# ModelSim

## XE

## User's Manual

**Version 5.3**

**The Ultimate Tool for VHDL or Verilog Simulation**

ModelSim XE is produced by Model Technology Incorporated. Unauthorized copying, duplication, or other reproduction is prohibited without the written consent of Model Technology.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Model Technology. The program described in this manual is furnished under a license agreement and may not be used or copied except in accordance with the terms of the agreement. The online documentation provided with this product may be printed by the end-user. The number or copies that may be printed is limited to the number of licenses purchased.

Model*Sim* is a trademark of Model Technology Incorporated. PostScript is a registered trademark of Adobe Systems Incorporated. UNIX is a registered trademark of AT&T in the USA and other countries. FLEXlm is a trademark of Globetrotter Software, Inc. IBM, AT, and PC are registered trademarks, AIX and RISC System/6000 are trademarks of International Business Machines Corporation. Windows, Microsoft, and MS-DOS are registered trademarks of Microsoft Corporation. OSF/Motif is a trademark of the Open Software Foundation, Inc. in the USA and other countries. SPARC is a registered trademark and SPARCstation is a trademark of SPARC International, Inc. Sun Microsystems is a registered trademark, and Sun, SunOS and OpenWindows are trademarks of Sun Microsystems, Inc. All other trademarks and registered trademarks are the properties of their respective holders.

Copyright (c) 1990 -1999, Model Technology Incorporated.
All rights reserved. Confidential. Online documentation may be printed by licensed customers of Model Technology Incorporated for internal business purposes only.

Software Version: 5.3d

Published: February 2000

**ModelSim XE support is available from:**

Xilinx, Inc.
2100 Logic Drive
San Jose, CA 95124-3400

phone: 800-255-7778
support email: hotline@xilinx.com
support web: http://support.xilinx.com

## Software License Agreement

IMPORTANT – USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS

CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE

This license is a legal “Agreement” concerning the use of Software between you, the end user, either individually or as an authorized representative of the company purchasing the license, and Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Mentor Graphics (Singapore) Private Limited, and their majority-owned subsidiaries (“Mentor Graphics”). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within 10 days after receipt of Software and receive a full refund of any license fee paid.

END USER LICENSE AGREEMENT

**1. GRANT OF LICENSE.** The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, and documentation (“Software”) are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics or its authorized distributor grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics’ then-current standard policies, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be communicated and technically implemented through the use of authorization codes or similar devices); (c) eligibility to receive updates, modifications, and revisions; and (d) support services provided. Current standard policies are available upon request.

**2. ESD SOFTWARE.** If you purchased a license to use embedded software development (“ESD”) Software, Mentor Graphics or its authorized distributor grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of

Mentor Graphics' real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

**3. BETA CODE.**

**3.1** Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may chose not to release commercially in any form.

**3.2** If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

**3.3** You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or makes during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

**4. RESTRICTIONS ON USE.** You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than your employer's employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by the European Union Software Directive or local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it without

Mentor Graphics' prior written consent. The provisions of this section shall survive the termination or expiration of this Agreement.

**5. LIMITED WARRANTY.**

**5.1** Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LOANED TO YOU FOR A LIMITED TERM OR AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

**5.2** THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

**6. LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE STATUTE OR REGULATION, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

**7. LIFE ENDANGERING ACTIVITIES.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN

CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH SUCH USE.

**8. INFRINGEMENT.**

**8.1** Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright in the United States, Canada, Japan, Switzerland, Norway, Israel, Egypt, or the European Union. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the claim, provided that you: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the claim; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the claim.

**8.2** If an infringement claim is made, Mentor Graphics may, at its option and expense, either (a) replace or modify Software so that it becomes noninfringing, or (b) procure for you the right to continue using Software. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

**8.3** Mentor Graphics has no liability to you if the alleged infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you design or market; (f) any Beta Code contained in Software; or (g) any Software provided by Mentor Graphics' licensors which do not provide such indemnification to Mentor Graphics' customers.

**8.4** THIS SECTION 8 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

**9. TERM.** This Agreement remains effective until expiration or termination. This Agreement will automatically terminate if you fail to comply with any term or condition of this Agreement or if you fail to pay for the license when due and such failure to pay continues for a period of 30 days after written notice from Mentor Graphics. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify

deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

**10. EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export in any manner any Software or direct product of Software, without first obtaining all necessary approval from appropriate local and United States government agencies.

**11. RESTRICTED RIGHTS NOTICE**. Software has been developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/ manufacturer is Mentor Graphics Corporation, 8005 Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

**12. THIRD PARTY BENEFICIARY**. For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth in this Agreement.

**13. CONTROLLING LAW.** This Agreement shall be governed by and construed under the laws of Ireland if the Software is licensed for use in Israel, Egypt, Switzerland, Norway, South Africa, or the European Union, the laws of Japan if the Software is licensed for use in Japan, the laws of Singapore if the Software is licensed for use in Singapore, People's Republic of China, Republic of China, India, or Korea, and the laws of the state of Oregon if the Software is licensed for use in the United States of America, Canada, Mexico, South America or anywhere else worldwide not provided for in this section.

**14. SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

**15. MISCELLANEOUS.** This Agreement contains the entire understanding between the parties relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement which are physically signed by you and an authorized agent of Mentor Graphics. This Agreement may only be modified by a physically signed writing between you and an authorized agent of Mentor Graphics. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and

expenses.

Rev. 10/99

# Table of Contents


Software License Agreement . . . 3

## 1 - Introduction (19)

ModelSim editions documented in this manual . . . 19
Standards supported . . . 20
SDF and ModelSim XE . . . 20
Assumptions . . . 20
Sections in this document . . . 21
Command reference . . . 22
Text conventions . . . 22
What is an "HDL item" . . . 23

## 2 - Design Libraries (25)

Design library contents . . . 26
Design unit information . . . 26
Design library types . . . 26
Library management commands . . . 27
Working with design libraries . . . 27
Creating a library . . . 27
Viewing and deleting library contents . . . 29
Assigning a logical name to a design library . . . 31
Moving a library . . . 33
Specifying the resource libraries . . . 34
VHDL resource libraries . . . 34
Predefined libraries . . . 34
Alternate IEEE libraries supplied . . . 35
Regenerating your design libraries . . . 35

## 3 - Projects and system initialization (37)

What is a project? . . . 38
A new file extension . . . 38
INI and MPF file comparison . . . 38
The [Project] section in the .mpf file . . . 39

Project operations . . . 39
Creating a Project . . . 40
Working with a Project . . . 43
Open a project . . . 43
Compile a project . . . 43
Simulating a project . . . 43
Modifying a project . . . 43
The project command . . . 44

## 4 - VHDL Simulation (45)

Compiling VHDL designs . . . 46
Creating a design library . . . 46
Invoking the VHDL compiler . . . 46
Dependency checking . . . 47
Simulating VHDL designs . . . 47
Invoking the simulator from the Main window . . . 47
Using the TextIO package . . . 49
Syntax for file declaration . . . 49
Using STD_INPUT and STD_OUTPUT within ModelSim . . . 50
TextIO implementation issues . . . 50
Writing strings and aggregates . . . 50
Reading and writing hexadecimal numbers . . . 51
Dangling pointers . . . 51
The ENDLINE function . . . 52
The ENDFILE function . . . 52
Using alternative input/output files . . . 52
Providing stimulus . . . 52
Obtaining the VITAL specification and source code . . . 53
VITAL packages . . . 53
ModelSim VITAL compliance . . . 53
VITAL compliance checking . . . 54
Compiling and Simulating with accelerated VITAL packages . . . 54

## 5 - Verilog Simulation (55)

ModelSim variables . . . 56

Compilation . . . 57
Incremental compilation . . . 58
Library usage . . . 60
Verilog-XL compatible compiler options . . . 62
Verilog-XL ‘uselib compiler directive . . . 65
Simulation . . . 67
Invoking the simulator . . . 67
Simulation resolution limit . . . 67
Event order issues . . . 68
Verilog-XL compatible simulator options . . . 70
Cell Libraries . . . 73
SDF timing annotation . . . 73
Delay modes . . . 73
System Tasks . . . 74
IEEE Std 1364-1995 system tasks . . . 75
Verilog-XL compatible system tasks . . . 77
Compiler Directives . . . 80
IEEE Std 1364-1995 compiler directives . . . 80
Verilog-XL compatible compiler directives . . . 81
Using the Verilog PLI . . . 82
Registering PLI applications . . . 82
Compiling and linking PLI applications . . . 84
The callback reason argument . . . 87
The sizetf callback function . . . 88
Object handles . . . 89
Third party PLI applications . . . 89
Support for VHDL objects . . . 90
IEEE Std 1364 ACC routines . . . 91
IEEE Std 1364 TF routines . . . 93
Verilog-XL compatible routines . . . 94
PLI tracing . . . 95

## 6 - Multiple logfiles, datasets and virtuals (97)

Multiple logfiles and datasets . . . 97
Opening and viewing datasets . . . 98
Using datasets with ModelSim commands . . . 100
Restricting the dataset prefix display . . . 101
Virtual Objects (User-defined buses, and more) . . . 102

Virtual signals . . . 103
Virtual functions . . . 104
Virtual regions . . . 104
Virtual types . . . 105
Logfile and virtual commands reference table . . . 105

## 7 - ModelSim XE Graphic Interface (107)

Window overview . . . 108
Window features . . . 109
Quick access toolbars . . . 110
Drag and Drop . . . 110
Command history . . . 111
Automatic window updating . . . 111
Finding names, and locating cursors . . . 112
Sorting HDL items . . . 112
Menu tear off . . . 112
Combine signals into a user-defined bus . . . 112
Tree window hierarchical view . . . 113
Main window . . . 115
The Main window menu bar . . . 116
The Main window tool bar . . . 121
The Main window status bar . . . 123
Mouse and keyboard shortcuts in the Transcript and Source windows . . . 123
Dataflow window . . . 126
The Dataflow window menu bar . . . 127
Tracing HDL items with the Dataflow window . . . 128
Saving the Dataflow window as a Postscript file . . . 129
List window . . . 130
HDL items you can view . . . 130
The List window menu bar . . . 131
Setting List window display properties . . . 133
Adding HDL items to the List window . . . 136
Editing and formatting HDL items in the List window . . . 137
Examining simulation results with the List window . . . 140
Finding items by name in the List window . . . 141
Setting time markers in the List window . . . 141
List window keyboard shortcuts . . . 142
Saving List window data to a file . . . 143

Process window . . . 144
The Process window menu bar . . . 145
Signals window . . . 146
The Signals window menu bar . . . 148
Selecting HDL item types to view . . . 149
Forcing signal and net values . . . 149
Adding HDL items to the Wave and List windows or a logfile . . . 151
Finding HDL items in the Signals window . . . 152
Defining clock signals . . . 153
Source window . . . 154
The Source window menu bar . . . 155
The Source window tool bar . . . 157
Editing the source file in the Source window . . . 158
Checking HDL item values and descriptions . . . 158
Setting Source window options . . . 159
Structure window . . . 160
The Structure window menu bar . . . 161
Variables window . . . 163
The Variables window menu bar . . . 164
Wave window . . . 166
Wave window panes . . . 169
HDL items you can view . . . 170
The Wave window menu bar . . . 171
Wave window tool bar . . . 174
Adding HDL items in the Wave window . . . 177
Combining and grouping items in the Wave window . . . 178
Editing and formatting HDL items in the Wave window . . . 180
Setting Wave window display properties . . . 184
Sorting a group of HDL items . . . 185
Finding items by name or value in the Wave window . . . 185
Using time cursors in the Wave window . . . 186
Zooming - changing the waveform display range . . . 188
Wave window keyboard shortcuts . . . 189
Saving waveforms . . . 191
Compiling with the graphic interface . . . 197
Locating source errors during compilation . . . 198
Setting default compile options . . . 198
Simulating with the graphic interface . . . 204
Design selection page . . . 205

VHDL settings page . . . 207
Verilog settings page . . . 209
SDF settings page . . . 211
Setting default simulation options . . . 213
ModelSim Quick Start . . . 216

## 8 - Standard Delay Format (SDF) Timing Annotation (219)

SDF and ModelSim XE . . . 220
Specifying SDF files for simulation . . . 220
Instance specification . . . 220
SDF specification with the GUI . . . 221
Errors and warnings . . . 221
VHDL VITAL SDF . . . 222
SDF to VHDL generic matching . . . 222
Resolving errors . . . 223
Verilog SDF . . . 224
The $sdf_annotate system task . . . 224
SDF to Verilog construct matching . . . 225
Optional edge specifications . . . 229
Optional conditions . . . 230
Rounded timing values . . . 231
SDF for Mixed VHDL and Verilog Designs . . . 231
Interconnect delays . . . 231
Troubleshooting . . . 232
Specifying the wrong instance . . . 232
Mistaking a component or module name for an instance label . . . 233
Forgetting to specify the instance . . . 233
Obtaining the SDF specification . . . 234

## 9 - Value Change Dump (VCD) Files (235)

ModelSim VCD commands and VCD tasks . . . 236
A VCD file from source to output . . . 236
VHDL source code . . . 236
VCD simulator commands . . . 237
VCD output . . . 237

## 10 - Tcl and ModelSim (241)


Tcl features within ModelSim . . . 242
Tcl References . . . 242
Tcl print references . . . 242
Tcl online references . . . 242
Tcl commands . . . 243
Tcl command syntax . . . 244
if command syntax . . . 247
set command syntax . . . 248
Command substitution . . . 248
Command separator . . . 249
Multiple-line commands . . . 249
Evaluation order . . . 249
Tcl relational expression evaluation . . . 250
Variable substitution . . . 250
System commands . . . 251
List processing . . . 251
VSIM Tcl commands . . . 252
ModelSim Tcl time commands . . . 252
Conversions . . . 253
Relations . . . 253
Arithmetic . . . 254


## A - ModelSim Variables (255)


Variable settings report . . . 256
Personal preferences . . . 256
Returning to the original ModelSim defaults . . . 256
Environment variables . . . 257
ModelSim Environment Variables . . . 257
Setting environment variables in Windows . . . 258
Referencing environment variables within ModelSim . . . 259
Removing temp files (VSOUT) . . . 260
Preference variables located in INI and MPF files . . . 261
[Library] library path variables . . . 261
[vcom] VHDL compiler control variables . . . 262
[vlog] Verilog compiler control variables . . . 264

[vsim] simulator control variables . . . 264
[Project] project file section (MPF files only) . . . 268
Setting variables in INI / MPF files . . . 269
Variable functions . . . 269
Preference variables located in TCL files . . . 273
User-defined variables . . . 273
Viewing the default preference file (pref.tcl) . . . 273
Preference variable arrays . . . 274
Main window preference variables . . . 276
The Main window uses preference variables similar to other ModelSim window to conrol colors and fonts. The variables below control some additional functions. . . . 276
Individual preference variables . . . 277
The addons variable . . . 277
Setting Tcl preference variables . . . 278
More preferences . . . 283
Preference variable loading order . . . 284
Simulator state variables . . . 285
Referencing simulator state variables . . . 285

B - ModelSim Shortcuts (287)

Wave window keyboard shortcuts . . . 287
List window keyboard shortcuts . . . 288
Command shortcuts . . . 288
Command history shortcuts . . . 289
Mouse and keyboard shortcuts in the Transcript and Source windows . . . 289
Right mouse button . . . 290

C - Tips and Techniques (291)

Running command-line and batch-mode simulations . . . 292
Passing parameters to macros . . . 293
Source code security and -nodebug . . . 294
Setting up libraries for group use . . . 294
Detecting infinite zero-delay loops . . . 295
Modeling memory in VHDL . . . 295

Index (301)

# 1 - Introduction

## Chapter contents

Standards supported . . . . . 18

Assumptions . . . . . 18

Sections in this document . . . . . 19

Text conventions. . . . . 20

What is an "HDL item" . . . . . 21

## Model*Sim* editions documented in this manual

**Model*Sim* XE Starter**
If you are new to HDL design, ModelSim XE Starter enables you to experiment with and create small VHDL and Verilog designs and test benches.

**Model*Sim* XE**
If you are using the Xilinx XC9500 CPLD and Spartan FPGA series of programmable logic devices or lower-density XC4000 and Virtex FPGAs, Model*Sim* XE provides a more powerful step into the world of HDL simulation. Model*Sim* XE supports behavioral, RTL, and gate-level simulation of Xilinx cell libraries for both Verilog and VHDL environments.

**Common Features**
All Model*Sim* tools, including both Xilinx versions, feature a complete HDL simulation and debugging environment providing 100 percent VHDL and Verilog language coverage, a source code viewer/editor, waveform viewer, design structure browser, list window and a host of other features designed to enhance productivity.

This documentation was written for Model*Sim* XE version 5.3d for Microsoft Windows NT 4.0 and Windows 95/98. If the Model*Sim* software you are using is a later release, check the README file that accompanied the software. Any supplemental information will be there. The online documentation included with Model*Sim* may be more current than the printed version as well.

## Standards supported

Model*Sim* VHDL supports both the IEEE 1076-1987 and 1076-1993 VHDL, 1164-1993 *Standard Multivalue Logic System for VHDL Interoperability*, and the 1076.2-1996 *Standard VHDL Mathematical Packages* standards. Any design developed with Model*Sim* will be compatible with any other VHDL system that is compliant with either IEEE Standard 1076-1987 or 1076-1993.

Model*Sim* Verilog is based on the IEEE Std 1364-1995 *Standard Hardware Description Language Based on the Verilog Hardware Description Language*. The Open Verilog International *Verilog LRM version 2.0* is also applicable to a large extent. Both PLI (Programming Language Interface) and VCD (Value Change Dump) are supported for ModelSim PE and EE users.

In addition, all products support SDF 1.0 through 3.0, VITAL 2.2b, and VITAL'95 - IEEE 1076.4-1995.

### SDF and ModelSim XE

For Model*Sim* XE, SDF timing annotation can only be applied to the Xilinx libraries shown below; all other libraries will simulate without annotation. The following mappings are defined in XE's *modelsim.ini* file:

```
simprim = $MODEL_TECH/../xilinx/vhdl/simprim
logiblox = $MODEL_TECH/../xilinx/vhdl/logiblox
unisim = $MODEL_TECH/../xilinx/vhdl/unisim
unisim5K = $MODEL_TECH/../xilinx/vhdl/unisim5K
simprims_ver = $MODEL_TECH/../xilinx/verilog/simprims
uni3000 = $MODEL_TECH/../xilinx/verilog/uni3000
unisims_ver = $MODEL_TECH/../xilinx/verilog/unisims_ver
uni5200 = $MODEL_TECH/../xilinx/verilog/uni5200
uni9000 = $MODEL_TECH/../xilinx/verilog/uni9000
```

## Assumptions

We assume that you are familiar with the use of your operating system. If you are not familiar with Microsoft Windows, we recommend that you work through the tutorials provided with MS Windows before using Model*Sim*.

We also assume that you have a working knowledge of VHDL and Verilog. Although Model*Sim* is an excellent tool to use while learning HDL concepts and practices, this document is not written to support that goal.

Finally, we make the assumption that you have worked the appropriate lessons in the *ModelSim Tutorial* and are therefore familiar with the basic functionality of Model*Sim*.

The *ModeSim Tutorial* is available from the Model*Sim* **Help** menu.

## Sections in this document

In addition to this introduction, you will find the following major sections in this document:

2 - Design Libraries (2-23)
To simulate an HDL design using Model*Sim*, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter.

3 - Projects and system initialization (3-35)
This chapter provides a definition of a Model*Sim* "project" and discusses the use of a new file extension for project files.

4 - VHDL Simulation (4-43)
This chapter is an overview of compilation and simulation for VHDL within the Model*Sim* environment.

5 - Verilog Simulation (5-53)
This chapter is an overview of compilation and simulation for Verilog within the Model*Sim* environment.

6 - Multiple logfiles, datasets and virtuals (6-95)
This chapter describes logfiles, datasets, and virtuals - new methods for viewing and organizing simulation data in Model*Sim*.

7 - ModelSim XE Graphic Interface (7-105)
This chapter describes the graphic interface available while operating VSIM, the Model*Sim* simulator. Model*Sim*'s graphic interface is designed to provide consistency throughout all operating system environments.

8 - Standard Delay Format (SDF) Timing Annotation (8-217)
This chapter discusses Model*Sim*'s implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

9 - Value Change Dump (VCD) Files (9-233)
This chapter explains Model Technology's Verilog VCD implementation for Model*Sim*. The VCD usage is extended to include VHDL designs.

10 - Tcl and ModelSim (10-239)
This chapter provides an overview of Tcl (tool command language) as used with Model*Sim*. Additional Tcl and Tk (Tcl's toolkit) can be found through several Tcl online references (10-240).

A - ModelSim Variables (A-253)
This appendix environment, system and preference variables used in Model*Sim*.

B - ModelSim Shortcuts (B-285)
A collection of Model*Sim* keyboard and mouse shortcuts.

C - Tips and Techniques (C-289)
An extended collection of Model*Sim* usage examples taken from our manuals, and tech support solutions.

## Command reference

The complete command reference for all Model*Sim* commands is located in the *ModelSim Command Reference*. Command Reference cross reference page numbers are prefixed with "CR", i.e.,"ModelSim Commands" (CR-11).

## Text conventions

Text conventions used in this manual include:

| | |
|---|---|
| *italic text* | provides emphasis and sets off filenames, path names, and design units names |
| **bold text** | indicates commands, command options, menu choices, package and library logical names, as well as variables and dialog box selection |
| `monospaced type` | monospace type is used for program and command examples |
| The right angle (>) | is used to connect menu choices when traversing menus as in: **File > Save** |
| UPPER CASE | denotes file types used by Model*Sim*, i.e., DO, WLF, INI, MPF, PDF, etc. |

## What is an "HDL item"

Because Model*Sim* works with both VHDL and Verilog, "HDL" refers to either VHDL or Verilog when a specific language reference is not needed. Depending on the context, "HDL item" can refer to any of the following:

| | |
|---|---|
| **VHDL** | block statement, component instantiation, constant, generate statement, generic, package, signal, or variable |
| **Verilog** | function, module instantiation, named fork, named begin, net, task, or register variable |

# 2 - Design Libraries

## Chapter contents


Design library contents . . . . . 24
Design unit information. . . . . 24
Design library types . . . . . 24
Library management commands. . . . . 25
Working with design libraries . . . . . 25
Creating a library . . . . . 25
Viewing and deleting library contents . . . . . 27
Assigning a logical name to a design library . . . . . 29
Moving a library . . . . . 31
Specifying the resource libraries . . . . . 32
VHDL resource libraries . . . . . 32
Predefined libraries . . . . . 32
Alternate IEEE libraries supplied . . . . . 33
Regenerating your design libraries . . . . . 33


VHDL has a concept of a *library*, which is an object that contains compiled design units; libraries are given names so they may be referenced. Verilog designs simulated within Model*Sim* are compiled into libraries as well. To simulate an HDL design using Model*Sim*, you need to know how to create, compile, maintain, and delete design libraries as described in this chapter. For additional information on Model*Sim* "Library management commands" (2-25) introduced in this chapter see "ModelSim Commands" (CR-11).

# Design library contents

A *design library* is a directory that serves as a repository for compiled design units. The design units contained in a design library consist of VHDL entities, packages, architectures, configurations, and Verilog modules and UDPs (user defined primitives). The design units are classed as follows:

- **Primary design units**
  Consists of entities, package declarations, configuration declarations, modules, and UDPs. Primary design units within a given library must have unique names.
- **Secondary design units**
  Consist of architecture bodies and package bodies. Secondary design units are associated with a primary design unit. Architectures by the same name can exist if they are associated with different entities.

## Design unit information

The information stored for each design unit in a design library is:

- retargetable, executable code
- debugging information
- dependency information

# Design library types

There are two kinds of design libraries: working libraries and resource libraries. A *working library* is the library into which a design unit is placed after compilation. A *resource library* contains design units that can be referenced within the design unit being compiled. Only one library can be the working library; in contrast, any number of libraries (including the working library itself) can be resource libraries during the compilation.

The library named **work** has special attributes within Model*Sim*; it is predefined in the compiler and need not be declared explicitly (i.e. **library work**). It is also the library name used by the compiler as the default destination of compiled design units. In other words the **work** library is the *working* library. In all other aspects it is the same as any other library.

## Library management commands

These library management commands are available from the Transcript window command line, or from the Model*Sim* graphic interface. Only brief descriptions are provided here; for more information and command syntax see "ModelSim Commands" (CR-11).

| Command | Description |
|---|---|
| vdel (CR-111) | deletes a design unit from a specified library |
| vlib (CR-140) | selectively lists the contents of a library. |
| vlib (CR-140) | creates a design library |
| vmake (CR-145) | outputs a UNIX-like makefile that can be used to reconstruct the library |
| vmap (CR-147) | defines or displays a mapping between a logical library name and a directory by modifying the *modelsim.ini* file |

These command may also be invoked from the Windows 95/98 or Windows NT DOS prompt.

## Working with design libraries

The implementation of a design library is not defined within standard VHDL or Verilog. Within Model*Sim*, design libraries are implemented as directories and can have any legal name allowed by the operating system, with one exception; extended identifiers are not supported for library names.

### Creating a library

Before you run the compiler, you need to create a working design library. This can be done from either the DOS command line or from the Model*Sim* graphic interface.

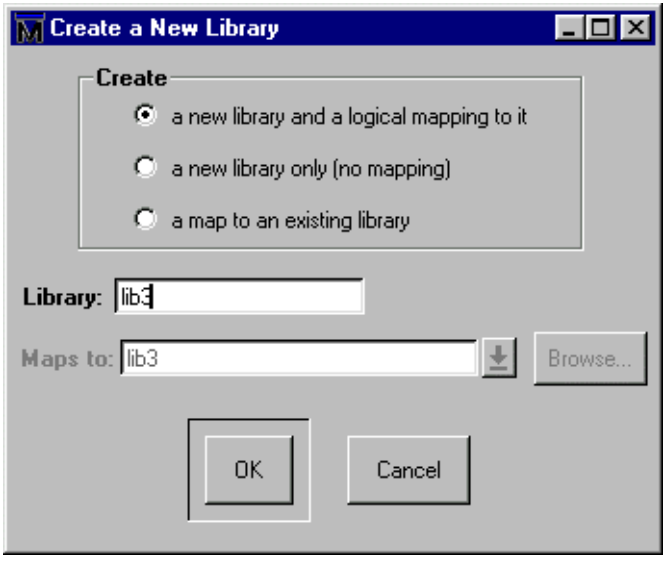
#### Creating a working library from the command line

From either the DOS prompt, or the Model*Sim* prompt, use this **vlib** command (CR-140):

```
vlib <directory_pathname>
```

### Creating a working library with the graphic interface

To create a new library with the Model*Sim* graphic interface, use the Transcriptwindow menu selection: **Design > Create a New Library**. This brings up a dialog box that allows you to specify the library name along with several mapping options.




The **Create a New Library** dialog box includes these options:

**Create**

- **a new library and a logical mapping to it**
  Type the new library name into the **Library** field. This creates a library sub-directory in your current working directory, initially mapped to itself. Once created, the mapped library is easily remapped to a different library.
- **a new library only (no mapping)**
  Type the new library name into the **Library** field. This creates a library sub-directory in your current working directory.
- **a map to an existing library**
  Type the new library name into the **Library** field, then type into the **Maps to** field or **Browse** to select a library name for the mapping.

and

- **Library**
  Type the new library name into this field.

- **Maps to**
  Type or **Browse** for a mapping for the specified library. This field can be changed only when the **create a map to an existing library** option is selected.

When you click **OK**, Model*Sim* creates the specified library directory and writes a specially-formatted file named *_info* into that directory. The *_info* file must remain in the directory to distinguish it as a Model*Sim* library.

If a mapping option is selected, a map entry is written to the *modelsim.ini* file in the [Library] section. See "[Library] library path variables" (A-259) for more information.

**Note:** It is important to remember that a design library is a special kind of directory; the only way to create a library is to use the Model*Sim* GUI, or the **vlib** command (CR-140). Do not create libraries using DOS or Windows.

## Viewing and deleting library contents

The contents of a design library can be viewed or deleted using either the command line or graphic interface.

### Viewing and deleting library contents from the command line

Use the **vlib** command (CR-140) to view the contents of a specified library (the contents of the **work** library are shown if no library is specified). Its syntax is:

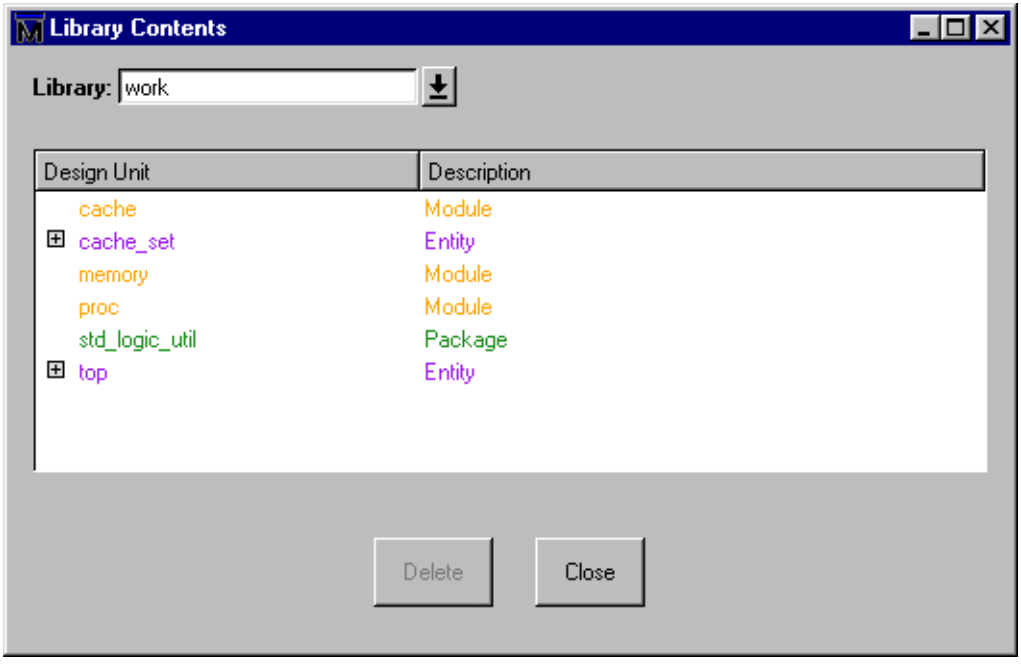
```
vdir -lib <library_name>
```

Use the **vdel** command (CR-111) to delete an entire library or a design unit from a specified library (the design unit is deleted from the **work** library if no library name is specified). Its syntax is:

```
vdel -lib <library_name> <design_unit>
```

### Viewing and deleting library contents with the graphic interface

Selecting **Design > View Library Contents…** allows you to view the design units (configurations, modules, packages, entities, and architectures) in the current library and delete selected design units.




The **Library Contents** dialog box includes these options:

- **Library**
  Select the library you wish to view from the drop-down list.
- **DesignUnit/Description list**

  Entity/architecture pairs are indicated by a box prefix; select a plus (+) box to view the associated architecture, or select a minus (–) box to hide the architecture.

  You can delete a package, configuration, or entity by selecting it and clicking **Delete**. This will remove the design unit from the library. If you delete an entity that has one or more architectures, the entity and all its associated architectures will be deleted.

You can also delete an architecture without deleting its associated entity. Just select the desired architecture name and click **Delete**. You are prompted for confirmation before any design unit is actually deleted.

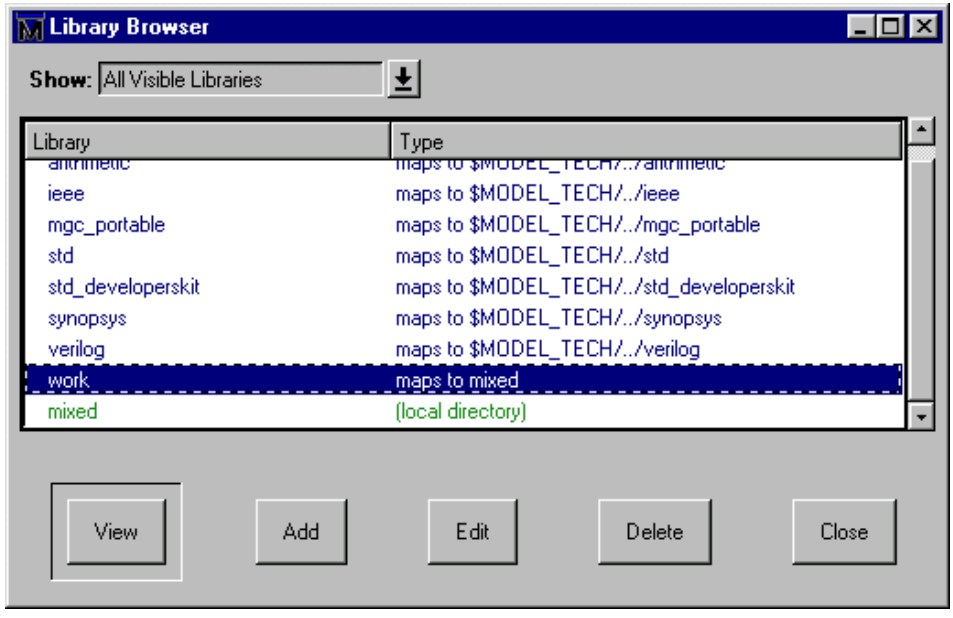
## Assigning a logical name to a design library

VHDL uses logical library names that can be mapped to Model*Sim* library directories. By default, Model*Sim* can find libraries in your current directory (assuming they have the right name), but for it to find libraries located elsewhere, you need to map a logical library name to the pathname of the library.

You can use the graphic interface, a command or the project file to assign a logical name to a design library.

### Library mappings with the GUI

To associate a logical name with a library, you select the **Design** > **Browse Libraries** command. This brings up a dialog box that allows you to view, add, edit, and delete mappings, as shown below:



The **Library Browser** dialog box includes these options:

- **Show**
  Choose the mapping and library scope to view from the drop-down list.
- **Library/Type list**

  *To view the contents of a library*
  Select the library, then click the **View** button. This brings up the **Library Contents** (2-27) dialog box. From there you can also delete design units from the library.

  *To create a new library mapping*
  Click the **Add** button. This brings up **Create a New Library** (2-25) dialog box that allows you to enter a new logical library name and the pathname to which it is to be mapped.

  It is possible to enter the name of a non-existent directory, but the specified directory must exist as a Model*Sim* library before you can compile design units into it. When you complete all your mapping changes and click the **OK** button in the Library Browser dialog box, Model*Sim* will issue a warning if any mappings are unresolved.

  *To edit an existing library mapping*
  Select the desired mapping entry, then click the **Edit** button. This brings up a dialog box that allows you to modify the logical library name and the pathname to which it is mapped. Selecting **Delete** removes an existing library mapping, but it does not delete the library. The library can be deleted with this **vdel** command (CR-111):

```
vdel -lib <library_name> -all
```

### Library mapping from the command line

You can issue a Model*Sim*/PLUS command to set the mapping between a logical library name and a directory; its form is:

```
vmap <logical_name> <directory_pathname>
```

This command may be invoked from either a DOS prompt or from the command line within Model*Sim*.

When you use vmap (CR-147) this way you are modifying the *modelsim.ini* file. You can also modify *modelsim.ini* manually by adding a mapping line. To do this, edit the *modelsim.ini* file using any text editor and add a line under the [Library] section heading using the syntax:

```
<logical_name> = <directory_pathname>
```

More than one logical name can be mapped to a single directory. For example, suppose the *modelsim.ini* file in the current working directory contains following lines:

```
[Library]
work = /usr/rick/design
my_asic = /usr/rick/design
```

This would allow you to use either the logical name **work** or **my_asic** in a **library** or **use** clause to refer to the same design library.

The **vmap** command (CR-147) can also be used to display the mapping of a logical library name to a directory. To do this, enter the shortened form of the command:

```
vmap <logical_name>
```

### Library search rules

The system searches for the mapping of a logical name in the following order:

- First the system looks for a *modelsim.ini* file.
- If the system doesn't find a *modelsim.ini* file, or if the specified logical name does not exist in the *modelsim.ini* file, the system searches the current working directory for a subdirectory that matches the logical name.

An error is generated by the compiler if you specify a logical name that does not resolve to an existing directory.

### See also

See "ModelSim Commands" (CR-11) for more information about the library management commands, "ModelSim XE Graphic Interface" (7-105) for more information about the graphical user interface, and "Projects and system initialization" (3-35) for more information about the *modelsim.ini* file.

## Moving a library

*Individual* design units in a design library cannot be moved. An *entire* design library can be moved, however, by using standard operating system commands for moving a directory.

# Specifying the resource libraries

## VHDL resource libraries

Within a VHDL source file, you can use the VHDL **library** clause to specify logical names of one or more resource libraries to be referenced in the subsequent design unit. The scope of a **library** clause includes the text region that starts immediately after the **library** clause and extends to the end of the declarative region of the associated design unit. *It does not extend to the next design unit in the file.*

Note that the **library** clause is not used to specify the working library into which the design unit is placed after compilation; the **vcom** (CR-106) adds compiled design units to the current working library. By default, this is the library named **work**. To change the current working library, you can use **vcom -work** and specify the name of the desired target library.

## Predefined libraries

Certain resource libraries are predefined in standard VHDL. The library named **std** contains the packages **standard** and **textio**, which should not be modified. The contents of these packages and other aspects of the predefined language environment are documented in the *IEEE Standard VHDL Language Reference Manual*, *Std 1076-1987* and *ANSI/IEEE Std 1076-1993*. See also, "Using the TextIO package" (4-47).

A VHDL **use** clause can be used to select specific declarations in a library or package that are to be visible within a design unit during compilation. A **use** clause references the compiled version of the package—not the source.

By default, every design unit is assumed to contain the following declarations:

```
LIBRARY std, work;
USE std.standard.all
```

To specify that all declarations in a library or package can be referenced, you can add the suffix *.all* to the library/package name. For example, the **use** clause above specifies that all declarations in the package **standard** in the design library named **std** are to be visible to the VHDL design file in which the **use** clause is placed. Other libraries or packages are not visible unless they are explicitly specified using a **library** or **use** clause.

Another predefined library is **work**, the library where a design unit is stored after it is compiled as described earlier. There is no limit to the number of libraries that can be referenced, but only one library is modified during compilation.

## Alternate IEEE libraries supplied

The installation directory may contain two or more versions of the IEEE library:

- *ieeepure*
  Contains only IEEE approved std_logic_1164 packages (accelerated for VSIM).
- *ieee*
  Contains precompiled Synopsys and IEEE arithmetic packages for the std_logic base type, which have been accelerated by Model Technology.

You can select which library to use by changing the mapping in the *modelsim.ini* file. The *modelsim.ini* file in the installation directory defaults to the *ieee* library.

## Regenerating your design libraries

Depending on your current Model*Sim* version, you may need to regenerate your design libraries before running a simulation. Check the installation README file to see if your libraries require an update. You can easily regenerate your design libraries with **-refresh**. You must use **vcom** (CR-106) with the **-refresh** option to update the VHDL design units in a library, and **vlog** (CR-141) with the **-refresh** option to update Verilog design units. By default, the work library is updated; use **-work <library>** to update a different library. For example, if you have a library named **mylib** that contains both VHDL and Verilog design units:

```
vcom -work mylib -refresh
vlog -work mylib -refresh
```

An important feature of **-refresh** is that it rebuilds the library image without using source code. This means that models delivered as compiled libraries without source code can be rebuilt for a specific release of Model*Sim* (4.6 and later only). In general, this works for moving forwards or backwards on a release. Moving backwards on a release may not work if the models used compiler switches or directives (Verilog only) that do not exist in the older release.

**Note:** As in the example above, you will need to use **vcom** for VHDL and **vlog** for Verilog design units. Also, you **don't** need to regenerate the std, ieee, vital22b, and verilog libraries. Also, you cannot use the **-refresh** option to update libraries that were built before the 4.6 release.

# 3 - Projects and system initialization

## Chapter contents


What is a project? . . . . . 36

A new file extension . . . . . 36

INI and MPF file comparison . . . . . 36

The [Project] section in the .mpf file . . . . . 37

Project operations . . . . . 37

Creating a Project . . . . . 38

Working with a Project . . . . . 41
- Open a project . . . . . 41
- Compile a project . . . . . 41
- Simulating a project . . . . . 41
- Modifying a project . . . . . 41
- The project command . . . . . 42


This chapter provides a definition of a Model*Sim* "project" and discusses the use of a new file extension for project files.

With the 5.3 release, Model*Sim* incorporates the file extension ***.mpf*** to denote project files. In past releases the *modelsim.ini* file (the system initialization file) was used as the project file.

**Note:** If you are looking for information on project file variables, please see *Appendix A - ModelSim Variables*.

## What is a project?

A project is a collection entity for an HDL design under specification or test. At a minimum, it has a root directory, a work library and session state which is stored in a .mpf file located in the project's root directory. A project may also consist of:

- HDL source files
- subdirectories
- Local libraries
- References to global libraries

## A new file extension

Why create a new file extension instead of using the .ini extension?

Project files with the new .mpf file extension contain some information that is specific to a given design and project directory. For this reason, use of a .mpf file is associated with current working directories residing in the project's directory tree.

On Windows systems the .ini file extension is used extensively to represent configuration files for many applications, including the OS. By changing the project file extension to .mpf, a new file type can be defined for Windows systems that won't be confused with other configuration files. Please note that old .ini files are still supported.

## INI and MPF file comparison

What is the difference between old project (.ini) files and new project (.mpf) files?

- A .ini file specifies initial tool settings and is fully supported outside of a project.
- By convention the new project files will have a .mpf extension.
- New features of the project file are most useful when used in conjunction with the Model*Sim* graphical user interface (GUI).
- A .mpf project file is specific to a given work session and may include the settings from a .ini file.
- A .mpf project file is located in the project working directory. This ensures that the path to a .mpf file will be <some_dir_path>/<project_name>/<project_name>.mpf

- A .mpf project file may be updated with current tool settings, whereas a .ini file is used for initial tool defaults. A .mpf project file also maintains changes to project settings.

## The [Project] section in the .mpf file

Sections within the .ini and .mpf files contain variable settings for libraries, the simulator, and compilers. The .mpf file includes an additional [Project] section located at the bottom of the file that contains one or more variables. These variables can be found in *Appendix A - ModelSim Variables*.

## Project operations

The Model*Sim* user has the ability to perform the following operations on a project:

Create: **File > New > New Project**

**New -** Inherit default settings from the current .ini file (must specify project name and working directory). Creates a fresh project file. Opens the new project.

**Copy -** Use an existing project, but change working directory. Copies all dependant files/libraries that are specified relative to the working directory. Absolute library paths are unchanged in the copied project file. Opens the new project.

Open: **File > Open > Open Project**

Open an existing project (change working directory, read settings from project file).

Edit: **Options > Edit Project**

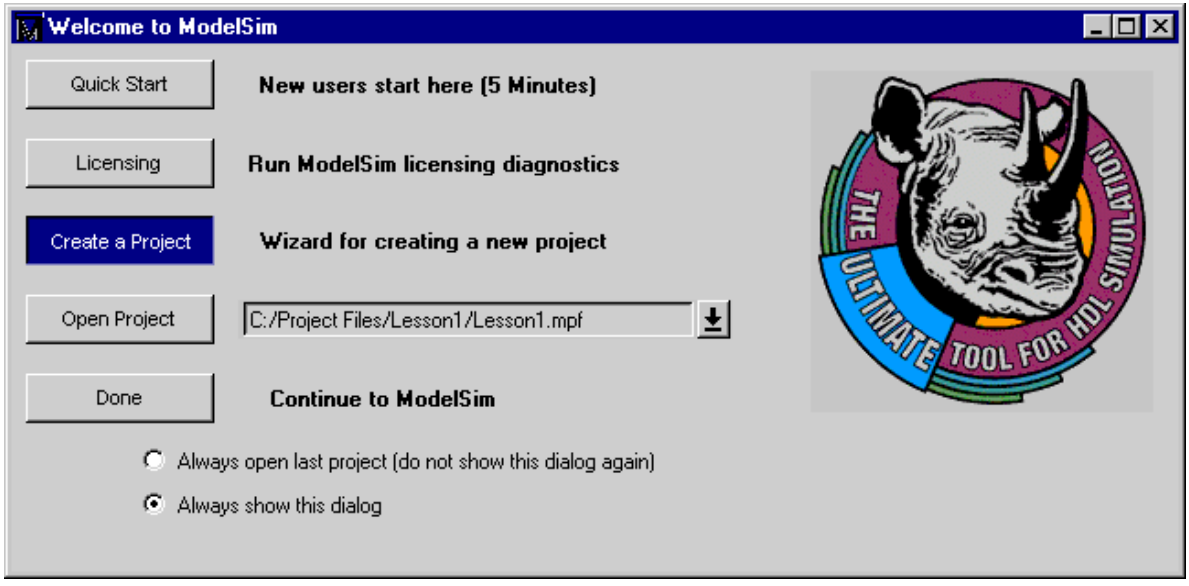
Allows you to edit an existing project (specify HDL source files, add and configure libraries, edit build scripts).

Delete: **File > Delete > Delete Project**

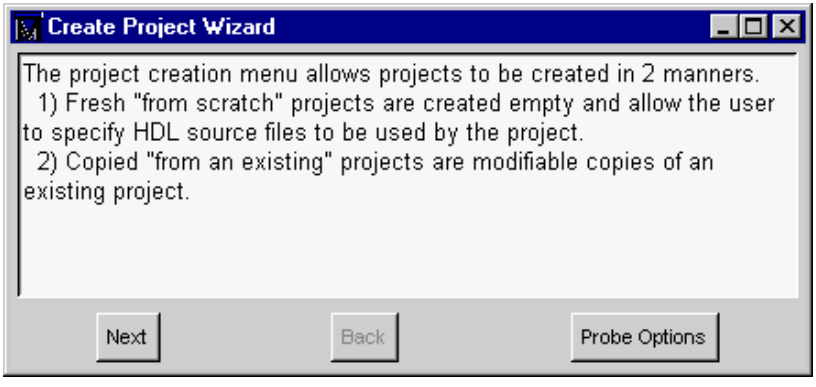
Deletes a specified project file. User must confirm relative file references and working directories. Absolute Library paths can also be optionally deleted.

## Creating a Project

1 To get started fast, select the **Create a Project** button from the Welcome to Model*Sim* screen that opens the first time you start Model*Sim* 5.3. If this screen is not available, you can enable it by selecting **Help > Enable Welcome** from the Main window.



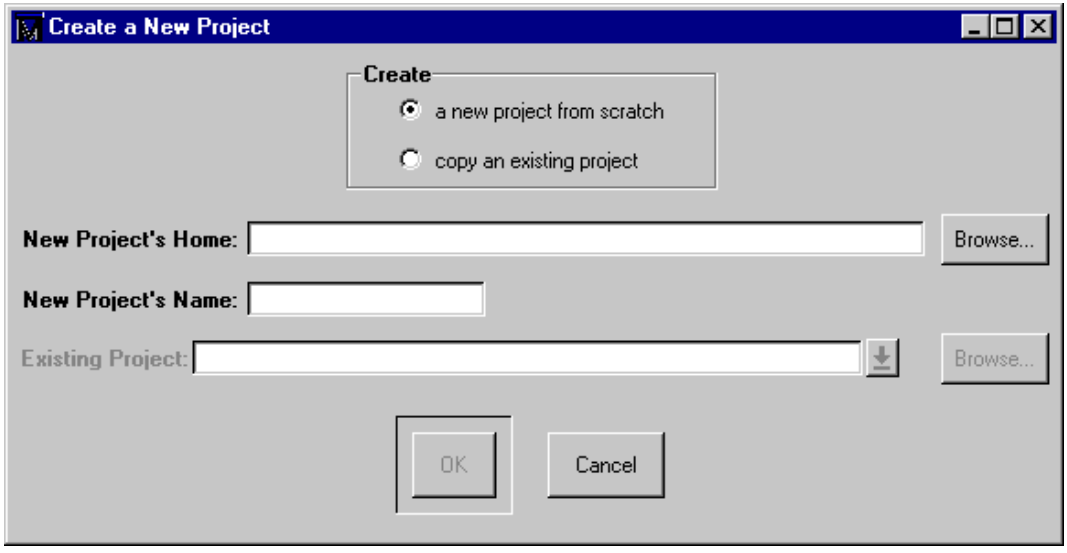

Clicking the **Create a Project** button opens the Create a New Project dialog box and a project creation wizard. The wizard guides you through each step of creating a new project, helping you create and load the project and providing the option of entering Verilog or VHDL descriptions.


Create Project Wizard
The project creation menu allows projects to be created in 2 manners.
1) Fresh "from scratch" projects are created empty and allow the user to specify HDL source files to be used by the project.
2) Copied "from an existing" projects are modifiable copies of an existing project.
Next
Back
Probe Options

**Note:** The Probe Options button allows you to probe the options within the Create a New Project dialog box. The Create Project Wizard displays option information as the cursor moves over each feature.

The Create a New Project dialog box can also be accessed by selecting **File > New > New Project** from the Model*Sim* Main window.

In the Create a New Project dialog box, you can elect to create a new project from scratch or copy an existing project.




If you select "create a new project from scratch," then:

2 Specify the "New Project's Home," which is the directory under which this project's directory tree will reside.

3 Specify the "New Project's Name," which will act as the project's directory name. It is recommended that a unique name be given to each project.

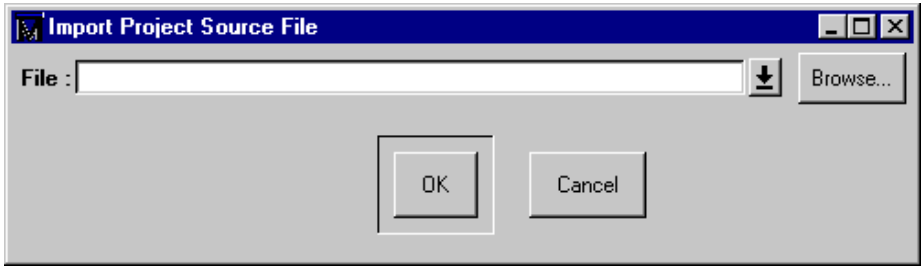
If you select "copy an existing project," then:

4 Specify an "Existing Project" name, which is the full path to an existing project's .mpf file.

**Note:** A project's MPF file is always located in the project's directory.

Once you have specified enough information for the project creation, the OK button is selectable. Select OK to create the project directory with a default work library, and open the project for use.

If you created the project from the "Welcome to ModelSim" dialog box, the project wizard now prompts you to enter an HDL source file. If you created the project using the **File > New > New Project** command, select **Options > Edit Project** to specify a source file.




Entry of an HDL source file name opens an editor session on the empty file. The source file must reside in the project's directory tree. When the editor session is complete you will be prompted to add the HDL source file to project's source list.

This completes the process of creating a project. The project will be open for use in the Main window. You can elect to leave Model*Sim* or edit the open project's HDL components until the project is completely specified and all files compile into libraries local to the project.

When you leave a Model*Sim* session, Model*Sim* will remember the last opened project so it can be reopened for your next session by simply clicking **Open Project** in the Welcome to Model*Sim* screen.

# Working with a Project

## Open a project

First, you must have a project open to work with it. To open a project select **File > Open > Open Project** from the Main window (cd'ing into projects directory won't work).

Once you have opened a project you can create HDL source files by selecting **File > New > New Source** from the Main window. When you create HDL files in the project's root directory you are prompted to add them to the project. HDL files for a given project must reside at or below the project's root directory.

## Compile a project

To compile your project's HDL description with the project open, select **Design > Compile Project** from the Main window, or click the Compile icon, and select the files you want to compile. Each file will be compiled into your project's work library. Click Done when you are finished.

## Simulating a project

To simulate an open project, select **Design > Load New Design** from the Main window or click the Load Design icon. On the Design tab of this menu you specify top level design unit for your project. On the VHDL and Verilog tabs you specify HDL specific simulator settings (these are described in the VSIM portion of the Reference Manual). On the SDF tab you can specify settings relating to the annotation of design timing from an SDF file (optional).

## Modifying a project

There are four types of project settings that can be modified; each is modified with a different action:

1. Project-wide settings describe the make up of the project. These settings are changed from the **Options > Edit Project** pull down menu.

2. Project compiler settings specify HDL compiler options. These settings are changed from the **Options > Compile** pull down menu.

3 Project design simulation settings describe how a specific design unit is loaded and simulated. The simulation settings are edited from the **Design > Load New Design** pull down menu or by clicking the Load Design icon.

4 Project simulation settings describe simulation specific behavior. These settings are edited from the **Options > Simulation** pull down menu.

Project setting changes take place at different times. General Project editing (**Options > Edit_Project**) is disabled while compiling or simulating and project-wide settings become effected after their change. Compiler option editing (**Options > Compile**) is disabled while compiling and takes effect at the next compile. Project design simulation settings (**Design > Load New Design**) take effect at design load/reload. Simulation option edits (**Options > Simulation**) are enabled during simulation and take effect immediately.

### Using project settings with the command line tools

Generally, projects are used only within the Model*Sim* graphical user interface. However, standalone tools will use the project file if they are invoked in the project's root directory. If invoked outside the project directory, the **MODELSIM** environment variable can be set with the path to the project file (*<Project_Root_Dir>/<Project_Name>.mpf*).

## The project command

The **project** command (CR-76) is used to perform common operations on new projects. The command is to be used outside of a simulation session. See "ModelSim Commands" (CR-11) for complete command details.

# 4 - VHDL Simulation

## Chapter contents


Compiling VHDL designs . . . . . 44
- Creating a design library . . . . . 44
- Invoking the VHDL compiler . . . . . 44
- Dependency checking . . . . . 45

Simulating VHDL designs . . . . . 45
- Invoking the simulator from the Main window . . . . . 45

Using the TextIO package . . . . . 47
- Syntax for file declaration . . . . . 47
- Using STD_INPUT and STD_OUTPUT within ModelSim . . . . . 48

TextIO implementation issues . . . . . 48
- Writing strings and aggregates . . . . . 48
- Reading and writing hexadecimal numbers . . . . . 49
- Dangling pointers . . . . . 49
- The ENDLINE function . . . . . 50
- The ENDFILE function . . . . . 50
- Using alternative input/output files . . . . . 50
- Providing stimulus . . . . . 50

Obtaining the VITAL specification and source code . . . . . 51

VITAL packages . . . . . 51

ModelSim VITAL compliance . . . . . 51
- VITAL compliance checking . . . . . 52
- Compiling and Simulating with accelerated VITAL packages . . . . . 52

Compiling and Simulating with accelerated VITAL packages . . . . . 52


This chapter provides an overview of compilation and simulation for VHDL designs within the Model*Sim*/PLUS environment, using the TextIO package with Model*Sim, and* Model*Sim*'s implementation of the VITAL (VHDL Initiative Towards ASIC Libraries) specification for ASIC modeling.

The TextIO package is defined within the *VHDL Language Reference Manuals, IEEE Std 1076-1987* and *IEEE Std 1076-1993*; it allows human-readable text input from a declared source within a VHDL file during simulation.

### Compiling and simulating with the GUI

Many of the examples in this chapter are shown from the command line. For compiling and simulation within Model*Sim*'s GUI see:

- Compiling with the graphic interface (7-195)
- Simulating with the graphic interface (7-202)

### Model*Sim* variables

Several variables are available to control simulation, provide simulator state feedback, or modify the appearance of the Model*Sim* GUI. To take effect, some variables, such as environment variables, must be set prior to simulation. See *Appendix A - ModelSim Variables* for a complete listing of Model*Sim* variables.

# Compiling VHDL designs

## Creating a design library

Before you can compile your design, you must create a library to store the compilation results. Use **vlib** (CR-140) to create a new library. For example:

```
vlib work
```

This creates a library named **work**. By default, compilation results are stored in the **work** library.

**Note:** The **work** library is actually a subdirectory named *work*. This subdirectory contains a special file named _info. Do not create libraries using MS Windows or DOS commands – always use the **vlib** command (CR-140).

See "Design Libraries" (2-23) for additional information on working with libraries.

## Invoking the VHDL compiler

Model*Sim* compiles one or more VHDL design units with a single invocation of **vcom** (CR-106), the VHDL compiler. The design units are compiled in the order that they appear on the command line. For VHDL, the order of compilation is important – you must compile any entities or configurations before an architecture that references them.

You can simulate a design containing units written with both the 1076 -1987 and 1076 -1993 versions of VHDL. To do so you will need to compile units from each

VHDL version separately. The **vcom** (CR-106) command compiles units written with version 1076 -1987 by default; use the -93 option with **vcom** (CR-106) to compile units written with version 1076 -1993. You can also change the default by modifying the *modelsim.ini* file (see *Chapter 3 - Projects and system initialization* for more information).

### Dependency checking

Dependent design units must be reanalyzed when the design units they depend on are changed in the library. **vcom** (CR-106) determines whether or not the compilation results have changed. For example, if you keep an entity and its architectures in the same source file and you modify only an architecture and recompile the source file, the entity compilation results will remain unchanged and you will not have to recompile design units that depend on the entity.

## Simulating VHDL designs

After compiling the design units, you can proceed to simulate your designs with **vsim** (CR-148). This section includes a discussion of simulation from the Windows/DOS command line. You can also use the graphic interface for simulation, see "Simulating with the graphic interface" (7-202).

**Note:** Simulation normally stops if a failure occurs, however, if a bounds check on a signal fails the simulator will continue running.

### Invoking the simulator from the Main window

For VHDL, invoke **vsim** (CR-148) with the name of the configuration, or entity/architecture pair. Note that if you specify a configuration you may not specify an architecture.

This example invokes **vsim** (CR-148) on the entity **my_asic** and the architecture **structure**:

```
vsim my_asic structure
```

If a design unit name is not specified, **vsim** (CR-148) will present the **Load Design** dialog box from which you can choose a configuration or entity/architecture pair. See "Simulating with the graphic interface" (7-202) for more information.

### Selecting the time resolution

The simulation time resolution is 1 ns by default. You can select a specific time resolution with the **vsim** (CR-148) **-t** option or from the **Load Design** dialog box. Available resolutions are: 1x, 10x or 100x of fs, ps, ns, us, ms, or sec.

For example, to run in picosecond resolution, or 10ps resolution respectively:

```
vsim -t ps topmod
vsim -t 10ps topmod
```

The default time resolution can also be changed by modifying the resolution (A-283) in the *modelsim.ini* file. You can view the current resolution by invoking the **report** command (CR-82) with the **simulator state** option.

See "Projects and system initialization" (3-35) for more information on modifying the *modelsim.ini* file.

**vsim** (CR-148) is capable of annotating a design using VITAL compliant models with timing data from an SDF file. You can specify the min:typ:max delay by invoking **vsim** (CR-148) with the **-sdfmin**, **-sdftyp** and **-sdfmax** options. Using the SDF file *f1.sdf* in the current work directory, the following invocation of **vsim** (CR-148) annotates maximum timing values for the design unit *my_asic*:

```
vsim -sdfmax /my_asic=f1.sdf
```

### Timing check disabling

By default, the timing checks within VITAL models are enabled. They are also disabled with the **+notimingchecks** option.

For example:

```
vsim +notimingchecks topmod
```

## Using the TextIO package

To access the routines in TextIO, include the following statement in your VHDL source code:

```
USE std.textio.all;
```

A simple example using the package TextIO is:

```
USE std.textio.all;
ENTITY simple_textio IS
END;

ARCHITECTURE simple_behavior OF simple_textio IS
BEGIN
    PROCESS
        VARIABLE i: INTEGER:= 42;
        VARIABLE LLL: LINE;
    BEGIN
        WRITE (LLL, i);
        WRITELINE (OUTPUT, LLL);
        WAIT;
    END PROCESS;
END simple_behavior;
```

### Syntax for file declaration

The VHDL'87 syntax for a file declaration is:

```
file identifier : subtype_indication is [ mode ]  file_logical_name ;
```

where "file_logical_name" must be a string expression.

The VHDL'93 syntax for a file declaration is:

```
file identifier_list : subtype_indication [ file_open_information ] ;
```

If a file is declared within an architecture, process, or package, the file is opened when you start the simulator and is closed when you exit from it. If a file is declared in a subprogram, the file is opened when the subprogram is called and closed when execution RETURNs from the subprogram.

You can specify a full or relative path as the file_logical_name; for example (VHDL'87):

```
file filename : TEXT is in "usr\rick\myfile";
```

### Using STD_INPUT and STD_OUTPUT within Model*Sim*

The standard VHDL'87 TextIO package contains the following file declarations:

```
file input: TEXT is in "STD_INPUT";
file output: TEXT is out "STD_OUTPUT";
```

The standard VHDL'93 TextIO package contains these file declarations:

```
file input: TEXT open read_mode is "STD_INPUT";
file output: TEXT open write_mode is "STD_OUTPUT";
```

STD_INPUT is a file_logical_name that refers to characters that are entered interactively from the keyboard, and STD_OUTPUT refers to text that is displayed on the screen.

In Model*Sim* reading from the STD_INPUT file brings up a dialog box that allows you to enter text into the current buffer. The last line written to the STD_OUTPUT file appears as a prompt in this dialog box. Any text that is written to the STD_OUTPUT file is also echoed in the Transcript window.

## TextIO implementation issues

### Writing strings and aggregates

A common error in VHDL source code occurs when a call to a WRITE procedure does not specify whether the argument is of type STRING or BIT_VECTOR. For example, the VHDL procedure:

```
WRITE (L, "hello");
```

will cause the following error:

```
ERROR: Subprogram "WRITE" is ambiguous.
```

In the TextIO package, the WRITE procedure is overloaded for the types STRING and BIT_VECTOR. These lines are reproduced here:

```
procedure WRITE(L: inout LINE; VALUE: in BIT_VECTOR;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

procedure WRITE(L: inout LINE; VALUE: in STRING;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

The error occurs because the argument "hello" could be interpreted as a string or a bit vector, but the compiler is not allowed to determine the argument type until it knows which function is being called.

The following procedure call also generates an error:

```
WRITE (L, "010101");
```

This call is even more ambiguous, because the compiler could not determine, even if allowed to, whether the argument "010101" should be interpreted as a string or a bit vector.

There are two possible solutions to this problem:

- Use a qualified expression to specify the type, as in:

```
WRITE (L, string'("hello"));
```

- Call a procedure that is not overloaded, as in:

```
WRITE_STRING (L, "hello");
```

The WRITE_STRING procedure simply defines the value to be a STRING and calls the WRITE procedure, but it serves as a shell around the WRITE procedure that solves the overloading problem. For further details, refer to the WRITE_STRING procedure in the io_utils package, which is located in the file *io_utils.vhd*.

## Reading and writing hexadecimal numbers

The reading and writing of hexadecimal numbers is not specified in standard VHDL. The Issues Screening and Analysis Committee of the VHDL Analysis and Standardization Group (ISAC-VASG) has specified that the TextIO package reads and writes only decimal numbers.

To expand this functionality, Model*Sim* supplies hexadecimal routines in the package io_utils, which is located in the file *io_utils.vhd*. To use these routines, compile the io_utils package and then include the following use clauses in your VHDL source code:

```
use std.textio.all;
use work.io_utils.all;
```

## Dangling pointers

Dangling pointers are easily incurred when using the TextIO package, because WRITELINE de-allocates the access type (pointer) that is passed to it. Following are examples of good and bad VHDL coding styles:

**Bad VHDL** (because L1 and L2 both point to the same buffer):

```
READLINE (infile, L1);    -- Read and allocate buffer
L2 := L1;                 -- Copy pointers
WRITELINE (outfile, L1);  -- Deallocate buffer
```

**Good VHDL** (because L1 and L2 point to different buffers):

```
READLINE (infile, L1);      -- Read and allocate buffer
L2 := new string'(L1.all);  -- Copy contents
WRITELINE (outfile, L1);    -- Deallocate buffer
```

## The ENDLINE function

The ENDLINE function described in the *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987* contains invalid VHDL syntax and cannot be implemented in VHDL. This is because access types must be passed as variables, but functions only allow constant parameters.

Based on an ISAC-VASG recommendation the ENDLINE function has been removed from the TextIO package. The following test may be substituted for this function:

```
(L = NULL) OR (L'LENGTH = 0)
```

## The ENDFILE function

In the *VHDL Language Reference Manuals, IEEE Std 1076-1987 and IEEE Std 1076-1993*, the ENDFILE function is listed as:

```
-- function ENDFILE (L: in TEXT) return BOOLEAN;
```

As you can see, this function is commented out of the standard TextIO package. This is because the ENDFILE function is implicitly declared, so it can be used with files of any type, not just files of type TEXT.

## Using alternative input/output files

You can use the TextIO package to read and write to your own files. To do this, just declare an input or output file of type TEXT.

The VHDL'87 declaration is:

```
file myinput : TEXT is in "pathname.dat";
```

For VHDL'93 the declaration is:

```
file myinput : TEXT open read_mode is "pathname.dat";
```

Then include the identifier for this file ("myinput" in this example) in the READLINE or WRITELINE procedure call.

## Providing stimulus

You can create batch files containing **force** (CR-56) commands that provide stimulus for simulation. A VHDL test bench has been included with the Model*Sim* install files as an example; it illustrates how results can be generated by reading vectors from a file. Check for this file:

<*install_dir*>*/examples/stimulus.vhd*

## Obtaining the VITAL specification and source code

### VITAL ASIC Modeling Specification

The IEEE 1076.4 VITAL ASIC Modeling Specification is available from the Institute of Electrical and Electronics Engineers, Inc.:

IEEE Customer Service
Hoes Lane
Tiscataway, NJ 08855-1331

Tel: (800)678-4333 ((908)562-5420 from outside the U.S.)
Fax: (908)981-9667
home page: http://www.ieee.org

### VITAL source code

The source code for VITAL packages is provided in the */<install_dir>/vhdl_src/vital22b*, or */vital95* directories.

## VITAL packages

VITAL v3.0 accelerated packages are pre-compiled into the **ieee** library in the installation directory.

**Note:** By default, Model*Sim* is optimized for VITAL v3.0. You can, however, revert to VITAL v2.2b by invoking **vsim** (CR-148) with the **-vital2.2b** option, and by mapping library **vital** to *<install_dir>/vital2.2b*.

## Model*Sim* VITAL compliance

A simulator is VITAL compliant if it implements the SDF mapping and if it correctly simulates designs using the VITAL packages, as outlined in the VITAL Model Development Specification. Model*Sim* VSIM is compliant with the IEEE 1076.4 VITAL ASIC Modeling Specification. In addition, Model*Sim* accelerates the VITAL_Timing and VITAL_Primitives packages. The procedures in these packages are optimized and built into the simulator kernel. By default, **vsim** (CR-148) uses the optimized procedures. The optimized procedures are functionally equivalent to the IEEE 1076.4 VITAL ASIC Modeling Specification (VITAL v3.0).

### VITAL compliance checking

If you are using VITAL 2.2b, you must turn off the compliance checking either by not setting the attributes, or by invoking **vcom** (CR-106) with the option **-novitalcheck**.

## Compiling and Simulating with accelerated VITAL packages

**vcom** (CR-106) automatically recognizes that a VITAL function is being referenced from the **ieee** library and generates code to call the optimized built-in routines.

Invoke  with the **-novital** option if you do not want to use the built-in VITAL routines. To exclude all VITAL functions, use **-novital all**:

```
vcom -novital all design.vhd
```

To exclude selected VITAL functions, use one or more **-novital <fname>** options:

```
vcom -novital VitalTimingCheck -novital VitalAND design.vhd
```

The **-novital** switch only affects calls to VITAL functions from the design units currently being compiled. Pre-compiled design units referenced from the current design units will still call the built-in functions unless they too are compiled with the **-novital** option.

Model*Sim* VITAL built-ins will be updated in step with new releases of the VITAL packages.

# 5 - Verilog Simulation

## Chapter contents


Compilation . . . . . 55
- Incremental compilation . . . . . 56
- Library usage . . . . . 58
- Verilog-XL compatible compiler options . . . . . 60
- Verilog-XL ‘uselib compiler directive . . . . . 63

Simulation . . . . . 65
- Invoking the simulator . . . . . 65
- Simulation resolution limit . . . . . 65
- Event order issues . . . . . 66
- Verilog-XL compatible simulator options . . . . . 68

Cell Libraries . . . . . 71
- SDF timing annotation . . . . . 71
- Delay modes . . . . . 71

System Tasks . . . . . 72
- IEEE Std 1364-1995 system tasks . . . . . 73
- Verilog-XL compatible system tasks . . . . . 75

Compiler Directives . . . . . 78
- IEEE Std 1364-1995 compiler directives . . . . . 78
- Verilog-XL compatible compiler directives . . . . . 79

Using the Verilog PLI . . . . . 80
- Registering PLI applications . . . . . 80
- Compiling and linking PLI applications . . . . . 82
- The callback reason argument . . . . . 85
- The sizetf callback function. . . . . 86
- Object handles . . . . . 87
- Third party PLI applications . . . . . 87
- Support for VHDL objects . . . . . 88
- IEEE Std 1364 ACC routines . . . . . 89
- IEEE Std 1364 TF routines . . . . . 91
- Verilog-XL compatible routines . . . . . 92
- PLI tracing . . . . . 93

This chapter describes how to compile and simulate Verilog designs with Model*Sim* Verilog. Model*Sim* Verilog implements the Verilog language as defined by the IEEE Std 1364-1995, and it is recommended that you obtain this specification as a reference manual.

In addition to the functionality described in the IEEE Std 1364-1995, Model*Sim* Verilog includes the following features:

- Standard Delay Format (SDF) annotator compatible with many ASIC and FPGA vendor's Verilog libraries.
- Value Change Dump (VCD) file extensions for ASIC vendor test tools.
- Dynamic loading of PLI applications.
- Compilation into retargetable, executable code.
- Incremental design compilation.
- Extensive support for mixing VHDL and Verilog in the same design (including SDF annotation).
- Graphic Interface that is common with Model*Sim* VHDL.
- Extensions to provide compatibility with Verilog-XL.

The following IEEE Std 1364-1995 functionality is not implemented in Model*Sim* Verilog:

- Array of instances (see section 7.1.5 in the IEEE Std 1364-1995).
- Verilog Procedural Interface (VPI) (see sections 22 and 23 in the IEEE Std 1364-1995).
- Macros (compiler `define directives) with arguments.

Many of the examples in this chapter are shown from the command line. For compiling and simulation within Model*Sim*'s GUI see:

- Compiling with the graphic interface (7-195)
- Simulating with the graphic interface (7-202)

## Model*Sim* variables

Several variables are available to control simulation, provide simulator state feedback, or modify the appearance of the Model*Sim* GUI. To take effect, some variables, such as environment variables, must be set prior to simulation. See *Appendix A - ModelSim Variables* for a complete listing of Model*Sim* variables.

## Compilation

Before you can simulate a Verilog design, you must first create a library and compile the Verilog source code into that library. This section provides detailed information on compiling Verilog designs. For information on creating a design library, see *Chapter 2 - Design Libraries*.

Model*Sim* Verilog compiles Verilog source code into retargetable, executable code, meaning that the library format is compatible across all supported platforms and that you can simulate your design on any platform without having to recompile your design specifically for that platform. As you compile your design, the resulting object code for modules and UDPs is generated into a library. By default, the compiler places results into the work library. You may specify an alternate library with the **-work** option. The following is a simple example of how to create a work library, compile a design, and simulate it:

Contents of top.v:

```
module top;
    initial $display("Hello world");
endmodule
```

Create the work library:

```
% vlib work
```

Compile the design:

```
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

View the contents of the work library (optional):

```
% vdir
MODULE top
```

Simulate the design:

```
% vsim -c top
# Loading work.top
VSIM 1> run -all
# Hello world
VSIM 2> quit
```

In this example, the simulator was run without the graphic interface by specifying the **-c** option. After the design was loaded, the simulator command **run -all** was entered, meaning to simulate until there are no more simulator events. Finally, the quit command was entered to exit the simulator. By default, a log of the simulation is written to the file "transcript" in the current directory.

## Incremental compilation

By default, Model*Sim* Verilog supports incremental compilation of designs, thus saving compilation time when you modify your design. Unlike other Verilog simulators, there is no requirement that you compile the entire design in one invocation of the compiler (although, you may do so if desired).

You are not required to compile your design in any particular order because all module and UDP instantiations and external hierarchical references are resolved when the design is loaded by the simulator. Incremental compilation is made possible by deferring these bindings, and as a result some errors cannot be detected during compilation. Commonly, these errors include: modules that were referenced but not compiled, incorrect port connections, and incorrect hierarchical references.

The following example shows how a hierarchical design can be compiled in top down order:

Contents of top.v:

```
module top;
    or2(n1, a, b);
    and2(n2, n1, c);
endmodule
```

Contents of and2.v:

```
module and2(y, a, b);
    output y;
    input a, b;
    and(y, a, b);
endmodule
```

Contents of or2.v:

```
module or2(y, a, b);
    output y;
    input a, b;
    or(y, a, b);
endmodule
```

Compile the design in top down order (assumes work library already exists):

```
% vlog top.v
-- Compiling module top

Top level modules:

    top
% vlog and2.v
-- Compiling module and2

Top level modules:

    and2
% vlog or2.v
-- Compiling module or2

Top level modules:

    or2
```

Note that the compiler lists each module as a top level module, although, ultimately, only "top" is a top level module. If a module is not referenced by another module compiled in the same invocation of the compiler, then it is listed as a top level module. This is just an informative message and can be ignored during incremental compilation. The message is more useful when you compile an entire design in one invocation of the compiler and need to know the top level module names for the simulator. For example,

```
% vlog top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

The most efficient method of incremental compilation is to manually compile only the modules that have changed. This is not always convenient, especially if your source files have compiler directive interdependencies (such as macros). In this case, you may prefer to always compile your entire design in one invocation of the compiler. If you specify the **-incr** option, the compiler will automatically determine which modules have changed and generate code only for those modules. This is not as efficient as manual incremental compilation because the compiler must scan all of the source code to determine which modules must be compiled.

The following is an example of how to compile a design with automatic incremental compilation:

```
% vlog -incr top.v and2.v or2.v
-- Compiling module top
-- Compiling module and2
-- Compiling module or2

Top level modules:
    top
```

Now, suppose that you modify the functionality of the "or2" module:

```
% vlog -incr top.v and2.v or2.v
-- Skipping module top
-- Skipping module and2
-- Compiling module or2

Top level modules:
    top
```

The compiler informs you that it skipped the modules "top" and "and2", and compiled "or2".

Automatic incremental compilation is intelligent about when to compile a module. For example, adding a comment to your source code does not result in a recompile; however, changing the compiler command line options results in a recompile of all modules.

## Library usage

All modules and UDPs in a Verilog design must be compiled into one or more libraries. One library is usually sufficient for a simple design, but you may want to organize your modules into various libraries for a complex design. If your design uses different modules having the same name, then you are required to put those modules in different libraries because design unit names must be unique within a library.

The following is an example of how you may organize your ASIC cells into one library and the rest of your design into another:

```
% vlib work
% vlib asiclib
% vlog -work asiclib and2.v or2.v
-- Compiling module and2
-- Compiling module or2
```

```
Top level modules:
    and2
    or2
% vlog top.v
-- Compiling module top

Top level modules:
    top
```

Note that the first compilation uses the **-work asiclib** option to instruct the compiler to place the results in the asiclib library rather than the default work library.

Since instantiation bindings are not determined at compile time, you must instruct the simulator to search your libraries when loading the design. The top level modules are loaded from the library named **work** unless you specify an alternate library with the **-lib** option. All other Verilog instantiations are resolved in the following order:

- Search libraries specified with **-Lf** options in the order they appear on the command line.
- Search the library specified in "Verilog-XL `uselib compiler directive" (5-63).
- Search libraries specified with **-L** options in the order they appear on the command line.
- Search the **work** library.
- Search the library explicitly named in the special escaped identifier instance name.

It is important to recognize that the work library is not necessarily a library named **work** - the **work** library refers to the library containing the module that instantiates the module or UDP that is currently being searched for. This definition is useful if you have hierarchical modules organized into separate libraries and if sub-module names overlap among the libraries. In this situation you want the modules to search for their sub-modules in the work library first. This is accomplished by specifying **-L work** first in the list of search libraries.

For example, assume you have a top level module "top" that instantiates module "modA" from library "libA" and module "modB" from library "libB". Furthermore, "modA" and "modB" both instantiate modules named "cellA", but the definition of "cellA" compiled into "libA" is different from that compiled into "libB". In this case, it is insufficient to just specify "-L libA - LlibB" as the search libraries because instantiations of "cellA" from "modB" resolve to the "libA" version of "cellA". The appropriate search library options are "-L work -L libA -L libB".

## Verilog-XL compatible compiler options

See **vlog** (CR-141) for a complete list of compiler options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to Model*Sim* Verilog.

`+define+<macro_name>[=<macro_text>]`
This option allows you to define a macro from the command line that is equivalent to the following compiler directive:

```
‘define <macro_name> <macro_text>
```

Multiple **+define** options are allowed on the command line. A command line macro overrides a macro of the same name defined with the ‘define compiler directive.

`+incdir+<directory>`
This option specifies directories to search for files included with **‘include** compiler directives. By default, the current directory is searched first and then the directories specified by the **+incdir** options in the order they appear on the command line. You may specify multiple **+incdir** options as well as multiple directories separated by "+" in a single **+incdir** option.

`+delay_mode_distributed`
This option disables path delays in favor of distributed delays. See Delay modes (5-71) for details.

`+delay_mode_path`
This option sets distributed delays to zero in favor of path delays. See Delay modes (5-71) for details.

`+delay_mode_unit`
This option sets path delays to zero and non-zero distributed delays to one time unit. See Delay modes (5-71) for details.

`+delay_mode_zero`
This option sets path delays and distributed delays to zero. See Delay modes (5-71) for details.

`-f <filename>`
This option reads more command line arguments from the specified text file. Nesting of **-f** options is allowed.

`+mindelays`
This option selects minimum delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

`+typdelays`
This option selects typical delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

`+maxdelays`
This option selects maximum delays from the "min:typ:max" expressions. If preferred, you may defer delay selection until simulation time by specifying the same option on the simulator.

`-u`
This option treats all identifiers in the source code as all uppercase.

### Options supporting source libraries

The following options support source libraries in the same manner as Verilog-XL. Note that these libraries are source libraries and are very different from the libraries that the Model*Sim* compiler uses to store compilation results. You may find it convenient to use these options if you are porting a design to Model*Sim* or if you are familiar with these options and prefer to use them.

Source libraries are searched after the source files on the command line are compiled. If there are any unresolved references to modules or UDPs, then the compiler searches the source libraries to satisfy them. The modules compiled from source libraries may in turn have additional unresolved references that cause the source libraries to be searched again. This process is repeated until all references are resolved or until no new unresolved references are found. Source libraries are searched in the order they appear on the command line.

`-v <filename>`
This option specifies a source library file containing module and UDP definitions. Modules and UDPs within the file are compiled only if they match previously unresolved references. Multiple **-v** options are allowed.

`-y <directory>`
This option specifies a source library directory containing module and UDP definitions. Files within this directory are compiled only if the file names match the names of previously unresolved references. Multiple **-y** options are allowed.

`+libext+<suffix>`

This option works in conjunction with the **-y** option. It specifies file extensions for the files in a source library directory. By default the compiler searches for files without extensions. If you specify the **+libext** option, then the compiler will search for a file with the suffix appended to an unresolved name. You may specify only one **+libext** option, but it may contain multiple suffixes separated by "+". The extensions are tried in the order they appear in the **+libext** option.

`+librescan`

This option changes how unresolved references are handled that are added while compiling a module or UDP from a source library. By default, the compiler attempts to resolve these references as it continues searching the source libraries. If you specify the **+librescan** option, then the new unresolved references are deferred until after the current pass through the source libraries. They are then resolved by searching the source libraries from the beginning in the order they are specified on the command line.

`+nolibcell`

By default, all modules compiled from a source library are treated as though they contain a **‘celldefine** compiler directive. This option disables this default. The **‘celldefine** directive only affects the PLI Access routines **acc_next_cell** and **acc_next_cell_load**.

`-R <simargs>`

This option instructs the compiler to invoke the simulator after compiling the design. The compiler automatically determines which top level modules are to be simulated. The command line arguments following **-R** are passed to the simulator, not the compiler. Place the **-R** option at the end of the command line or terminate the simulator command line arguments with a single "-" character to differentiate them from compiler command line arguments.

The **-R** option is not a Verilog-XL option, but it is used by Model*Sim* Verilog to combine the compile and simulate phases together as you may be used to with Verilog-XL. It is not recommended that you regularly use this option because you will incur the unnecessary overhead of compiling your design for each simulation run. Mainly, it is provided to ease the transition to Model*Sim* Verilog.

## Verilog-XL ‘uselib compiler directive

The **‘uselib** compiler directive is an alternative source library management scheme to the **-v**, **-y**, and **+libext** compiler options. It has the advantage that a design may reference different modules having the same name. The **‘uselib** compiler directive is not defined in the IEEE Std 1364-1995, but Model*Sim* supports it for compatibility with Verilog-XL.

The syntax for the **‘uselib** directive is:

```
‘uselib <library_reference>...
```

where <library_reference> is:

```
dir=<library_directory> | file=<library_file> | libext=<file_extension> |
lib=<library_name>
```

In Verilog-XL, the library references are equivalent to command line options as follows:

```
dir=<library_directory>    -y <library_directory>
file=<library_file>        -v <library_file>
libext=<file_extension>    +libext+<file_extension>
```

For example, the following directive

```
‘uselib dir=/h/vendorA libext=.v
```

is equivalent to the following command line options:

```
-y /h/vendorA +libext+.v
```

Since the **‘uselib** directives are embedded in the Verilog source code, there is more flexibility in defining the source libraries for the instantiations in the design. The appearance of a ‘uselib directive in the source code explicitly defines how instantiations that follow it are resolved, completely overriding any previous **‘uselib** directives.

For example, the following code fragment shows how two different modules that have the same name can be instantiated within the same design:

```
‘uselib dir=/h/vendorA file=.v
NAND2 u1(n1, n2, n3);
‘uselib dir=/h/vendorB file=.v
NAND2 u2(n4, n5, n6);
```

This allows the NAND2 module to have different definitions in the vendorA and vendorB libraries.

ModelSim Verilog supports the **‘uselib** directive in a different manner than Verilog-XL. The library files referenced by the **‘uselib** directive are not automatically compiled by ModelSim Verilog. The reason for this is that an object library is not allowed to contain multiple modules having the same name, and the results of a single invocation of the compiler can be placed in only one object library. Because it is an important feature of **‘uselib** to allow a design to reference multiple modules having the same name, independent compilation of the source libraries referenced by the **‘uselib** directives is required. Each source library should be compiled into its own object library. The compilation of the code containing the **‘uselib** directives only records which object libraries to search for each module instantiation when the design is loaded by the simulator.

Because the **‘uselib** directive is intended to reference source libraries, ModelSim Verilog must infer the object libraries from the library references. The rule is to assume an object library named **work** in the directory defined in the library reference **dir=<library_directory>** or the directory containing the file in the library reference **file=<library_file>**. The library reference **libext=<file_extension>** is ignored. For example, the following **‘uselib** directives infer the same object library:

```
‘uselib dir=/h/vendorA

‘uselib file=/h/vendorA/libcells.v
```

In both cases ModelSim Verilog assumes that the library source is compiled into the object library **/h/vendorA/work**.

ModelSim Verilog also extends the **‘uselib** directive to explicitly specify the object library with the library reference **lib=<library_name>**. For example,

```
‘uselib lib=/h/vendorA/work
```

The library name can be a complete path to a library, or it can be a logical library name defined with the **vmap** command. Since this usage of **‘uselib** is an extension, it may be desirable to qualify it with an **‘ifdef** to make it portable to other Verilog systems. For example,

```
‘ifdef MODEL_TECH
‘uselib lib=vendorA
‘else
‘uselib dir=/h/vendorA libext=.v
‘endif
```

The MODEL_TECH macro is automatically defined by the ModelSim compiler.

# Simulation

The Model*Sim* simulator can load and simulate both Verilog and VHDL designs, providing a uniform graphic interface and simulation control commands for debugging and analyzing your designs. The graphic interface and simulator commands are described elsewhere in this manual, while this section focuses specifically on Verilog simulation.

## Invoking the simulator

A Verilog design is ready for simulation after it has been compiled into one or more libraries. The simulator may then be invoked with the names of the top level modules (many designs contain only one top level module). For example, if your top level modules are "testbench" and "globals", then invoke the simulator as follows:

```
vsim testbench globals
```

If a top-level module name is not specified, VSIM will present the **Load Design** dialog box from which you can choose one or more top-level modules. See "Simulating with the graphic interface" (7-202) for more information.

After the simulator loads the top level modules, it iteratively loads the instantiated modules and UDPs in the design hierarchy, linking the design together by connecting up the ports and resolving hierarchical references. By default, all modules and UDPs are loaded from the library named **work**.

On successful loading of the design, the simulation time is set to zero, and you must enter a **run** command to begin simulation. Commonly, you enter **run -all** to run until there are no more simulation events or until **$finish** is executed in the Verilog code. You may also run for specific time periods, i.e., **run 100 ns**. Enter the **quit** command to exit the simulator.

## Simulation resolution limit

The simulator internally represents time as a 64-bit integer in units equivalent to the smallest unit of simulation time, also known as the simulation resolution limit. The resolution limit defaults to the smallest time precision found among all of the ‘timescale compiler directives in the design. The time precision is the second number in the ‘timescale directive. For example, "10 ps" in the following:

```
‘timescale 1 ns / 10 ps
```

The time precision should not be unnecessarily small because it will limit the maximum simulation time limit, and it will degrade performance in some cases. If the design contains no ‘timescale directives, then the resolution limit defaults to the "resolution" value specified in the *modelsim.ini* file (default is 1 ns). In any case, you may override the default resolution limit by specifying the -t option on the command line.

For example, to explicitly choose 100 ps resolution:

```
vsim -t 100ps top
```

This forces 100 ps resolution even if the design has finer time precision. As a result, time values with finer precision are rounded to the nearest 100 ps.

## Event order issues

The Verilog language is defined such that the simulator is not required to execute simultaneous events in any particular order. Unfortunately, some models are inadvertently written to rely on a particular event order, and these models may behave differently when ported to another Verilog simulator. A model with event order dependencies is ambiguous and should be corrected. For example, the following code is ambiguous:

```
module top;
    reg r;

    initial r = 0;
    initial r = 1;

    initial #10 $display(r);
endmodule
```

The value displayed for "r" depends on the order that the simulator executes the initial constructs that assign to "r". Conceptually, the initial constructs run concurrently and the simulator is allowed to execute them in any order. Model*Sim* Verilog executes the initial constructs in the order they appear in the module, and the value displayed for "r" is "1". Verilog-XL produces the same result, but a simulator that displays "0" is not incorrect because the code is ambiguous.

Since many models have been developed on Verilog-XL, Model*Sim* Verilog duplicates Verilog-XL event ordering as much as possible to ease the porting of those models to Model*Sim* Verilog. However, Model*Sim* Verilog does not match Verilog-XL event ordering in all cases, and if a model ported to Model*Sim* Verilog does not behave as expected, then you should suspect that there are event order dependencies.

Tracking down event order dependencies is a tedious task, so Model*Sim* Verilog aids you with a couple of compiler options:

`-compat`
This option turns off optimizations that result in different event ordering than Verilog-XL. Model*Sim* Verilog generally duplicates Verilog-XL event ordering, but there are caseswhere it is inefficient to do so. Using this option does not help you find the event order dependencies, but it allows you to ignore them. Keep in mind that this option does not account for all event order discrepancies, and that using this option may degrade performance.

`-hazards`
This option detects event order hazards involving simultaneous reading and writing of the same register in concurrently executing processes. To enable hazard detection you must invoke **vlog** (CR-141) with the **-hazards** option when you compile your source code and you must also invoke **vsim** (CR-148) with the **-hazards** option when you simulate.

The **vsim** command (CR-148) detects the following kinds of hazards:

- WRITE/WRITE:
  Two processes writing to the same variable at the same time.
- READ/WRITE:
  One process reading a variable at the same time it is being written to by another process. VSIM calls this a READ/WRITE hazard if it executed the read first.
- WRITE/READ:
  Same as a READ/WRITE hazard except that VSIM executed the write first.

The **vsim** command (CR-148) issues an error message when it detects a hazard. The message pinpoints the variable and the two processes involved. You can have the simulator break on the statement where the hazard is detected by setting the **break on assertion** level to **error**.

To enable hazard detection you must invoke **vlog** (CR-141) with the **-hazards** option when you compile your source code and you must also invoke **vsim** (CR-148) with the **-hazards** option when you simulate.

Limitations of hazard detection:

- Reads and writes involving bit and part selects of vectors are not considered for hazard detection. The overhead of tracking the overlap between the bit and part selects is too high.
- A WRITE/WRITE hazard is flagged even if the same value is written by both processes.

- A WRITE/READ or READ/WRITE hazard is flagged even if the write does not modify the variable's value.
- Glitches on nets caused by non-guaranteed event ordering are not detected.

## Verilog-XL compatible simulator options

See **vsim** (CR-148) for a complete list of simulator options. The options described here are equivalent to Verilog-XL options. Many of these are provided to ease the porting of a design to Model*Sim* Verilog.

`+alt_path_delays`

Specify path delays operate in inertial mode by default. In inertial mode, a pending output transition is cancelled before a new output transition is scheduled, and the delay is selected based on the transition from the current value of the net to the new pending value. The result is that an output may have no more than one pending transition at a time, and that pulses narrower than the delay are filtered. The +alt_path_delays option modifies the inertial mode such that a delay is based on a transition from a pending output value rather than the current value of the net. This option has no effect in transport mode (see **+pulse_e** (5-69) and **+pulse_r** (5-70)).

`-l <filename>`

By default, the simulation log is written to the file "transcript". The **-l** option allows you to specify an alternate file.

`+maxdelays`

This option selects the maximum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+mindelays`

This option selects the minimum value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

`+no_notifier`

This option disables the toggling of the notifier register argument of the timing check system tasks. By default, the notifier is toggled when there is a timing check violation, and the notifier usually causes a UDP to propagate an X. Therefore, the **+no_notifier** option suppresses X propagation on timing violations.

`+no_pulse_msg`

This option disables the warning message for specify path pulse errors. A path pulse error occurs when a pulse propagated through a path delay falls between the pulse rejection limit and pulse error limits set with the **+pulse_r** and **+pulse_e** options. A path pulse error results in a warning message, and the pulse is propagated as an X. The **+no_pulse_msg** option disables the warning message, but the X is still propagated.

`+no_tchk_msg`

This option disables error messages issued by timing check system tasks when timing check violations occur. However, notifier registers are still toggled any may result in the propagation of X's for timing check violations.

`+nosdfwarn`

This option disables warning messages during SDF annotation.

`+notimingchecks`

This option completely disables all timing check system tasks.

`+nowarn<mnemonic>`

This option disables the class of warning messages specified by <mnemonic>. This option only disables warning messages accompanied by a mnemonic enclosed in square brackets. For example,

```
# WARNING: test.v(2): [TFMPC] - Too few port connections.
```

This warning message can be disabled with the **+nowarnTFMPC** option.

`+pulse_e/<percent>`

This option controls how pulses are propagated through specify path delays, where <percent> is a number between 0 and 100 that specifies the error limit as a percentage of the path delay. A pulse greater than or equal to the error limit propagates to the output in transport mode (transport mode allows multiple pending transitions on an output). A pulse less than the error limit and greater than or equal to the rejection limit (see **+pulse_r** (5-70)) propagates to the output as an X. If the rejection limit is not specified, then it defaults to the error limit. For example, consider a path delay of 10 along with a **+pulse_e/80** option. The error limit is 80% of 10 and the rejection limit defaults to 80% of 10. This results in the propagation of pulses greater than or equal to 8, while all other pulses are filtered. Note that you can forcespecify path delays to operate in transport mode by using the **+pulse_e/0** option.

`+pulse_r/<percent>`
This option controls how pulses are propagated through specify path delays, where <percent> is a number between 0 and 100 that specifies the rejection limit as a percentage of the path delay. A pulse less than the rejection limit is suppressed from propagating to the output. If the error limit is not specified (see **+pulse_e** (5-69)), then it defaults to the rejection limit.

`+pulse_e_style_ondetect`
This option selects the "on detect" style of propagating pulse errors (see **+pulse_e** (5-69)). A pulse error propagates to the output as an X, and the "on detect" style is to schedule the X immediately, as soon as it has been detected that a pulse error has occurred. The "on event" style is the default for propagating pulse errors (see **+pulse_e_style_onevent** (5-70)).

`+pulse_e_style_onevent`
This option selects the "on event" style of propagating pulse errors (see **+pulse_e** (5-69)). A pulse error propagates to the output as an X, and the "on event" style is to schedule the X to occur at the same time and for the same duration that the pulse would have occurred if it had propagated through normally. The "on event" style is the default for propagating pulse errors.

`+sdf_nocheck_celltype`
By default, the SDF annotator checks that the CELLTYPE name in the SDF file matches the module or primitive name for the CELL instance. It is an error if the names do not match. The **+sdf_nocheck_celltype** option disables this error check.

`+sdf_verbose`
This option displays a summary of the design objects annotated for each SDF file.

`+transport_path_delays`
By default, path delays operate in inertial mode (pulses smaller than the delay are filtered). The **+transport_path_delays** option selects transport mode for path delays. In transport mode, narrow pulses are propagated through path delays. Note that this option affects path delays only, and not primitives. Primitives always operate in inertial delay mode.

`+typdelays`
This option selects the typical value in min:typ:max expressions. The default is the typical value. This option has no effect if the min:typ:max selection was determined at compile time.

# Cell Libraries

Model Technology is the first Verilog simulation vendor to pass the ASIC Council's Verilog test suite and achieve the "Library Tested and Approved" designation from Si2 Labs. This test suite is designed to ensure Verilog timing accuracy and functionality and is the first significant hurdle to complete on the way to achieving full ASIC vendor support. As a consequence, many ASIC and FPGA vendors' Verilog cell libraries are compatible with Model*Sim* Verilog.

The cell models generally contain Verilog "specify blocks" that describe the path delays and timing constraints for the cells. See section 13 in the IEEE Std 1364-1995 for details on specify blocks, and section 14.5 for details on timing constraints. Model*Sim* Verilog fully implements specify blocks and timing constraints as defined in the IEEE Std 1364-1995 along with some Verilog-XL compatible extensions.

## SDF timing annotation

Model*Sim* Verilog supports timing annotation from Standard Delay Format (SDF) files. See *Chapter 8 - Standard Delay Format (SDF) Timing Annotation* for details.

## Delay modes

Verilog models may contain both distributed delays and path delays. The delays on primitives, UDPs, and continuous assignments are the distributed delays, whereas the port-to-port delays specified in specify blocks are the path delays. These delays interact to determine the actual delay observed. Most Verilog cells use path delays exclusively, with the distributed delays set to zero. For example,

```
module and2(y, a, b);
    input a, b;
    output y;

    and(y, a, b);

    specify
        (a => y) = 5;
        (b => y) = 5;
    endspecify
endmodule
```

In the above two-input "and" gate cell, the distributed delay for the "and" primitive is zero, and the actual delays observed on the module ports are taken from the path delays. This is typical for most cells, but a complex cell may require non-zero

distributed delays to work properly. Even so, these delays are usually small enough that the path delays take priority over the distributed delays. The rule is that if a module contains both path delays and distributed delays, then the larger of the two delays for each path shall be used (as defined by the IEEE Std 1364-1995). This is the default behavior, but you can specify alternate delay modes with compiler directives and options. These options and directives are compatible with Verilog-XL. Compiler delay mode options take precedent over delay mode directives in the source code.

### Distributed delay mode

In distributed delay mode the specify path delays are ignored in favor of the distributed delays. Select this delay mode with the **+delay_mode_distributed** compiler option or the **‘delay_mode_distributed** compiler directive.

### Path delay mode

In path delay mode the distributed delays are set to zero. Select this delay mode with the **+delay_mode_path** compiler option or the **‘delay_mode_path** compiler directive.

### Unit delay mode

In unit delay mode the distributed delays are set to one (the unit is the time_unit specified in the **‘timescale** directive), and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_unit** compiler option or the **‘delay_mode_unit** compiler directive.

### Zero delay mode

In zero delay mode the distributed delays are set to zero, and the specify path delays and timing constraints are ignored. Select this delay mode with the **+delay_mode_zero** compiler option or the **‘delay_mode_zero** compiler directive.

## System Tasks

The IEEE Std 1364-1995 defines many system tasks as part of the Verilog language, and Model*Sim* Verilog supports all of these along with several non-standard Verilog-XL system tasks. The system tasks listed in this chapter are built into the simulator, although some designs depend on user-defined system tasks implemented with the Programming Language Interface (PLI). If the simulator issues warnings regarding undefined system tasks, then it is likely that these system tasks are defined by a PLI application that must be loaded by the simulator.

## IEEE Std 1364-1995 system tasks

The following system tasks are described in detail in the IEEE Std 1364-1995.

| Timescale tasks | Simulator control tasks | Simulation time functions |
|---|---|---|
| $printtimescale | $finish | $realtime |
| $timeformat | $timeformat | $stime |
| | | $time |

| Probabilistic distribution functions | Value change dump (VCD) file tasks | Conversion functions for reals |
|---|---|---|
| $dist_chi_square | $dumpall | $bitstoreal |
| $dist_erlang | $dumpfile | $itor |
| $dist_exponential | $dumpflush | $realtobits |
| $dist_normal | $dumplimit | $rtoi |
| $dist_poisson | $dumpoff | |
| $dist_t | $dumpon | |
| $dist_uniform | $dumpvars | |

| Display tasks | File I/O tasks | PLA modeling tasks |
|---|---|---|
| $display | $fclose | $async$and$array |
| $displayb | $fdisplay | $async$nand$array |
| $displayh | $fdisplayb | $async$or$array |
| $displayo | $fdisplayh | $async$nor$array |
| $monitor | $fdisplayo | $async$and$plane |
| $monitorb | $fmonitor | $async$nand$plane |

**Display tasks**

$monitorh
$monitoro
$monitoroff
$monitoron
$strobe
$strobeb
$strobeh
$strobeo
$write
$writeb
$writeh
$writeo

**File I/O tasks**

$fmonitorb
$fmonitorh
$fmonitoro
$fopen
$fstrobe
$fstrobeb
$fstrobeh
$fstrobeo
$fwrite
$fwriteb
$fwriteh
$fwriteo
$readmemb
$readmemh

**PLA modeling tasks**

$async$or$plane
$async$nor$plane
$sync$and$array
$sync$nand$array
$sync$or$array
$sync$nor$array
$sync$and$plane
$sync$nand$plane
$sync$or$plane
$sync$nor$plane

**Timing check task**

$hold
$nocharge
$period
$recovery
$setup
$setuphold
$skew
$width

**Stochastic analysis tasks**

$q_add
$q_exam
$q_full
$q_initialize
$q_remove
$random

## Verilog-XL compatible system tasks

The following system tasks are provided for compatibility with Verilog-XL. Although they are not part of the IEEE standard, they are described in an annex of the IEEE Std 1364-1995.

```
$countdrivers
$getpattern
$sreadmemb
$sreadmemh
```

The following system tasks are also provided for compatibility withVerilog-XL, but they are not described in the IEEE Std 1364-1995.

```
$sdf_annotate
```

See: The $sdf_annotate system task (8-222)

```
$test$plusargs("plus argument")
```

This system function tests for the presence of a specific plus argument on the simulator's command line. It returns 1 if the plus argument is present; otherwise, it returns 0. For example, to test for **+verbose**:

```
if ($test$plusargs("verbose"))
    $display("Executing cycle 1");
```

```
$removal(reference_event, data_event, limit, [notifier])
```

The $removal timing check issues a timing violation under the following condition:

0 < ((time of reference event) - (time of data event)) < limit

```
$recrem(reference_event, data_event, recovery_limit, removal_limit, [notifier],
    [tstamp_cond], [tcheck_cond], {delayed-reference], [delayed_data])
```

The $recrem timing check is a combined $recovery and $removal timing check. It behaves very much like the $setuphold timing check, along with the extensions for negative constraints and an alternate method of conditioning (see the description of $setuphold below)

The following system tasks are extended to provide additional functionality for negative timing constraints and an alternate method of conditioning, as does Verilog-XL.

```
$setuphold(clk_event, data_event, setup_limit, hold_limit, [notifier],
    [tstamp_cond], [tcheck_cond], [delayed_clk], [delayed_data])
```

The tstamp_cond argument conditions the data_event for the setup check and the clk_event for the hold check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The tcheck_cond argument conditions the data_event for the hold check and the clk_event for the setup check. This alternate method of conditioning precludes specifying conditions in the clk_event and data_event arguments.

The delayed_clk argument is a net that is continuously assigned the value of the net specified in the clk_event. The delay is non-zero if the setup_limit is negative, zero otherwise.

The delayed_data argument is a net that is continuously assigned the value of the net specified in the data_event. The delay is non-zero if the hold_limit is negative, zero otherwise.

The delayed_clk and delayed_data arguments are provided to ease the modeling of devices that may have negative timing constraints. The model's logic should reference the delayed_clk and delayed_data nets in place of the normal clk and data nets. This ensures that the correct data is latched in the presence of negative constraints. The simulator automatically calculates the delays for delayed_clk and delayed_data such that the correct data is latched as long as a timing constraint has not been violated.

```
$recovery(reference event, data_event, removal_limit, recovery_limit,
    [notifier], [tstamp_cond], [tcheck_cond], [delayed_reference],
    [delayed_data])
```

The $recovery system task normally takes a recovery_limit as the third argument and an optional notifier as the fourth argument. By specifying a limit for both the third and fourth arguments, the $recovery timing check is transformed into a combination removal and recovery timing check similar to the $recrem timing check. The only difference is that the removal_limit and recovery_limit are swapped.

The following system tasks are Verilog-XL system tasks that are not implemented in Model*Sim* Verilog, but have equivalent simulator commands.

```
$input("filename")
```

This system task read command test from the specified filename. The equivalent simulator command is **do <filename>**.

```
$list[(hierarchical_name)]
```

This system task lists the source code for the specified scope. The equivalent functionality is provided by selecting a module in the graphic interface Structure window. The corresponding source code is displayed in the source window.

```
$reset
```

This system task resets the simulation back to its time 0 state. The equivalent simulator command is **restart**.

```
$restart("filename")
```

This system task sets the simulation to the state specified by filename, saved in a previous call to $save. The equivalent simulator command is **restore <filename>**.

```
$save("filename")
```

This system task saves the current simulation state to the file specified by filename. The equivalent simulator command is **checkpoint <filename>**.

```
$scope(hierarchical_name)
```

This system task sets the interactive scope to the scope specified by hierarchical_name. The equivalent simulator command is **environment <pathname>**.

```
$showscopes
```

This system task displays a list of scopes defined in the current interactive scope. The equivalent simulator command is **show**.

```
$showvars
```

This system task displays a list of registers and nets defined in the current interactive scope. The equivalent simulator command is **show**.

# Compiler Directives

Model*Sim* Verilog supports all of the compiler directives defined in the IEEE Std 1364-1995 and some additional Verilog-XL compiler directives for compatibility.

Many of the compiler directives (such as **‘define** and **‘timescale**) take effect at the point they are defined in the source code and stay in effect until the directive is redefined or until it is reset to its default by a **‘resetall** directive. The effect of compiler directives spans source files, so the order of source files on the compilation command line may be significant. For example, if you have a file that defines some common macros for the entire design, then you may need to place it first in the list of files to be compiled.

The **‘resetall** directive affects only the following directives by resetting them back to their default settings (this information is not provided in the IEEE Std 1364-1995):

```
‘celldefine
‘define_nettype
‘delay_mode_distributed
‘delay_mode_path
‘delay_mode_unit
‘delay_mode_zero
‘timescale
‘unconnected_drive
‘uselib
```

Model*Sim* Verilog implicitly defines the following macro:

```
‘define MODEL_TECH
```

## IEEE Std 1364-1995 compiler directives

The following compiler directives are described in detail in the IEEE Std 1364-1995; however, the **‘define** directive is not fully implemented as described - it does not support macro arguments.

```
‘celldefine
‘default_nettype
‘define
‘else
‘endcelldefine
‘endif
‘ifdef
‘include
‘nounconnected_drive
‘resetall
‘timescale
```

```
‘unconnected_drive
‘undef
```

## Verilog-XL compatible compiler directives

The following compiler directives are provided for compatibility with Verilog-XL.

‘delay_mode_distributed

This directive disables path delays in favor of distributed delays. See Delay modes (5-71) for details.

‘delay_mode_path

This directive sets distributed delays to zero in favor of path delays. See Delay modes (5-71) for details.

‘delay_mode_unit

This directive sets path delays to zero and non-zero distributed delays to one time unit. See Delay modes (5-71) for details.

‘delay_mode_zero

This directive sets path delays and distributed delays to zero. See Delay modes (5-71) for details.

‘uselib

This directive is an alternative to the **-v**, **-y**, and **+libext** source library compiler options. See Verilog-XL ‘uselib compiler directive (5-63) for details.

The following Verilog-XL compiler directives are silently ignored by Model*Sim* Verilog. Many of these directives are irrelevant to Model*Sim* Verilog, but may appear in code being ported from Verilog-XL.

```
‘accelerate
‘autoexpand_vectornets
‘disable_portfaults
‘enable_portfaults
‘endprotect
‘expand_vectornets
‘noaccelerate
‘noexpand_vectornets
‘noremove_gatenames
‘noremove_netnames
‘nosuppress_faults
```

```
‘protect
‘remove_gatenames
‘remove_netnames
‘suppress_faults
```

The following Verilog-XL compiler directives produce warning messages in Model*Sim* Verilog. These are not implemented in Model*Sim* Verilog, and any code containing these directives may behave differently in Model*Sim* Verilog than in Verilog-XL.

```
‘default_decay_time
‘default_trireg_strength
‘signed
‘unsigned
```

# Using the Verilog PLI

The Verilog PLI provides a mechanism for defining system tasks and functions that communicate with the simulator through a C procedural interface. There are many third party applications available that interface to Verilog simulators through the PLI interface (see **Third party PLI applications** (5-87)). In addition, you may write your own PLI applications.

Model*Sim* Verilog implements the PLI as defined in the IEEE Std 1364, with the exception of the VPI routines and the acc_handle_datapath routine. Currently, the VPI routines are not commonly used, although Model*Sim* will support them in a future release. The acc_handle_datapath routine is not implemented because the information it returns is more appropriate for a static timing analysis tool.

The IEEE Std 1364 is the reference that defines the usage of the PLI routines. This manual only describes details of using the PLI with Model*Sim* Verilog.

## Registering PLI applications

Each PLI application must register its system tasks and functions with the simulator, providing the name of each system task and function and the associated callback routines. Since many PLI applications already interface to Verilog-XL, Model*Sim* Verilog PLI applications make use of the same mechanism to register information about each system task and function in an array of s_tfcell structures. This structure is declared in the veriuser.h include file as follows:

```
typedef int (*p_tffn)();

typedef struct t_tfcell {
    short type;/* USERTASK, USERFUNCTION, or USERREALFUNCTION */
    short data;/* passed as data argument of callback function */
```

```
        p_tffn checktf;  /* argument checking callback function */
        p_tffn sizetf;   /* function return size callback function */
        p_tffn calltf;   /* task or function call callback function */
        p_tffn misctf;   /* miscellaneous reason callback function */
    char *tfname;/* name of system task or function */

        /* The following fields are ignored by ModelSim Verilog */
    int forwref;
    char *tfveritool;
    char *tferrmessage;
    int hash;
    struct t_tfcell *left_p;
    struct t_tfcell *right_p;
    char *namecell_p;
    int warning_printed;
} s_tfcell, *p_tfcell;
```

The various callback functions (checktf, sizetf, calltf, and misctf) are described in detail in Section 17 of the IEEE Std 1364. The simulator calls these functions for various reasons. All callback functions are optional, but most applications contain at least the calltf function, which is called when the system task or function is executed in the Verilog code. The first argument to the callback functions is the value supplied in the data field (many PLI applications don't use this field). The type field defines the entry as either a system task (USERTASK) or a system function that returns either a register (USERFUNCTION) or a real (USERREALFUNCTION). The tfname field is the system task or function name (it must begin with $). The remaining fields are not used by Model*Sim* Verilog.

On loading of a PLI application, the simulator first looks for an init_usertfs function, and then a veriusertfs array. If init_usertfs is found, the simulator calls that function so that it can call mti_RegisterUserTF for each system task or function defined. The mti_RegisterUserTF function is declared in veriuser.h as follows:

```
void mti_RegisterUserTF(p_tfcell usertf);
```

The storage for each usertf entry passed to the simulator must persist throughout the simulation, because the simulator de-references the usertf pointer to call the callback functions. It is recommended that you define your entries in an array, with the last entry set to 0. If the array is named veriusertfs (as is the case for linking to Verilog-XL), then you don't have to provide an init_usertfs function, and the simulator will automatically register the entries directly from the array (the last entry must be 0). For example,

```
s_tfcell veriusertfs[] = {
    {usertask, 0, 0, 0, abc_calltf, 0, "$abc"},
    {usertask, 0, 0, 0, xyz_calltf, 0, "$xyz"},
```

```
    {0}  /* last entry must be 0 */
};
```

Alternatively, you can add an init_usertfs function to explicitly register each entry from the array:

```
void init_usertfs()
{
    p_tfcell usertf = veriusertfs;
    while (usertf->type)
        mti_RegisterUserTF(usertf++);
}
```

It is an error if the PLI application does not contain a veriusertfs array or an init_usertfs function.

Since PLI applications are dynamically loaded by the simulator, you must specify which applications to load (each application must be a dynamically loadable library, see Compiling and linking PLI applications). The PLI applications are specified as follows:

- As a list in the Veriuser entry in the *modelsim.ini* file:

```
Veriuser = pliapp1.so pliapp2.so pliappn.so
```

- As a list in the PLIOBJS environment variable:

```
% setenv PLIOBJS "pliapp1.so pliapp2.so pliappn.so"
```

- As a -pli option to the simulator (multiple options are allowed):

```
-pli pliapp1.so -pli pliapp2.so -pli pliappn.so
```

The various methods of specifying PLI applications may be used simultaneously.

## Compiling and linking PLI applications

Model*Sim* Verilog uses operating system calls to dynamically load PLI applications when the simulator loads a design. Therefore, the PLI application must be compiled and linked for dynamic loading on a specific operating system. The PLI routines are declared in the include files located in the Model*Sim* <*install_dir*>*/modeltech/include* directory. The acc_user.h file declares the ACC routines (defined in Section 19 of the IEEE Std 1364) and the veriuser.h file declares the TF routines (defined in Section 21 of the IEEE Std 1364).

The following instructions assume that the PLI application is in a single source file. For multiple source files, compile each file as specified in the instructions and link all of the resulting object files together with the specified link instruction.

### PLI Application Requirements

PLI applications are dynamically loaded into VSIM. A PLI application can consist of one or more dynamically loadable objects. Each of these objects must contain an entry point named init_usertfs( ) and a local veriusertfs table of user tasks and functions. There must be an entry in the table for each function in the object file that can be called externally. The init_usertfs( ) function must call mti_RegisterUserTF( ) for each entry in its local veriusertfs table.

PLI applications that use the TF functions should include the veriuser.h file.

### Windows NT/95/98 platforms

Under Windows NT/95/98, VSIM loads a 32-bit dynamically linked library for each PLI application. The following compile and link steps are used to create the necessary.dll file (and other supporting files) using the Microsoft Visual C/C++ compiler.

```
cl -c -I<install_dir>\modeltech\include app.c
    link -dll -export:<C_init_function> app.obj \
    <install_dir>\modeltech\win32\mtipli.lib
```

Where <C_init_function> is the function name specified in the FOREIGN attribute (for FLI).

**Note:** The PLI interface has been tested with DLLs built using Microsoft Visual C/C++ compiler version 4.1 or greater.

### Specifying the PLI file to load

Once your C application has been compiled it is ready to be loaded by VSIM. The name of the file to be loaded is specified in the *modelsim.ini* file by the Veriuser entry. The Veriuser entry must be in the [vsim] section of the file.

For example,

```
[vsim]
.
.
.
Veriuser = pliapp.dll
```

The Veriuser entry also accepts a list of shared objects. Each shared object is an independent PLI application that must contain an init_usertfs( ) entry point that registers the application's tasks and callback functions. An example entry in the *modelsim.ini* file is:

```
Veriuser = pliapp1.dll pliapp2.dll pliapp3.dll
```

See also *Appendix A - ModelSim Variables* for more information on the *modelsim.ini* file.

### PLI Example

The following example is a trivial, but complete PLI application.

```
hello.c:

    #include "veriuser.h"
    static hello()
    {
        io_printf("Hi there\n");
    }
    s_tfcell veriusertfs[] = {
        {usertask, 0, 0, 0, hello, 0, "$hello"},
        {0}  /* last entry must be 0 */
    };

hello.v:

    module hello;
        initial $hello;
    endmodule

Compile the PLI code for Solaris operating system:
    % cc -c -I<install_dir>/modeltech/include hello.c
    % ld -G -o hello.sl hello.o

Compile the Verilog code:
    % vlib work
    % vlog hello.v

Simulate the design:

    % vsim -c -pli hello.sl hello
    # Loading work.hello
    # Loading ./hello.sl
    VSIM 1> run -all
    # Hi there
    VSIM 2> quit
```

## The callback reason argument

The second argument to a callback function is the reason argument. The values of the various reason constants are defined in the veriuser.h include file. See Section 17 of the IEEE Std 1364 for a description of the reason constants. The following details relate to Model*Sim* Verilog, and may not be obvious in the IEEE Std 1364. Specifically, the simulator passes the reason values to the misctf callback functions under the following circumstances:

`reason_endofcompile`
For the completion of loading the design.

`reason_finish`
For the execution of the $finish system task or the quit command.

`reason_startofsave`
For the start of execution of the checkpoint command, but before any of the simulation state has been saved. This allows the PLI application to prepare for the save, but it shouldn't save its data with calls to tf_write_save until it is called with reason_save.

`reason_save`
For the execution of the checkpoint command. This is when the PLI application must save its state with calls to tf_write_save.

`reason_startofrestart`
For the start of execution of the restore command, but before any of the simulation state has been restored. This allows the PLI application to prepare for the restore, but it shouldn't restore its date with calls to tf_read_restart until it is called with reason_restart. The reason_startofrestart value is passed only for a restore command, and not in the case that the simulator is invoked with -restore.

`reason_restart`
For the execution of the restore command. This is when the PLI application must restore its state with calls to tf_read_restart.

`reason_reset`
For the execution of the restart command. This is when the PLI application should free its memory and reset its state.

`reason_endofreset`
For the completion of the restart command, after the simulation state has been reset but before the design has been reloaded.

`reason_interactive`
For the execution of the $stop system task or any other time the simulation is interrupted and waiting for user input.

`reason_scope`
For the execution of the environment command or selecting a scope in the structure window. Also for the call to acc_set_interactive_scope if the callback_flag argument is non-zero.

`reason_paramvc`
For the change of value on the system task or function argument.

`reason_synch`
For the end of time step event scheduled by tf_synchronize.

`reason_rosynch`
For the end of time step event scheduled by tf_rosynchronize.

`reason_reactivate`
For the simulation event scheduled by tf_setdelay.

`reason_paramdrc`
Not supported in Model*Sim* Verilog.

`reason_force`
Not supported in Model*Sim* Verilog.

`reason_release`
Not supported in Model*Sim* Verilog.

`reason_disable`
Not supported in Model*Sim* Verilog.

## The sizetf callback function

A user-defined system function specifies the width of its return value with the sizetf callback function, and the simulator calls this function while loading the design. The following details on the sizetf callback function are not found in the IEEE Std 1364:

- If you omits the sizetf function, then a return width of 32 is assumed.

- The sizetf function should return 0 if the system function return value is of Verilog type "real".
- The sizetf function should return -32 if the system function return value is of Verilog type "integer".

## Object handles

Many of the object handles returned by the ACC PLI routines are pointers to objects that naturally exist in the simulation data structures, and the handles to these objects are valid throughout the simulation, even after the acc_close routine is called. However, some of the objects are created on demand, and the handles to these objects become invalid after acc_close is called. The following object types are created on demand in Model*Sim* Verilog:

```
accOperator (acc_handle_condition)
accWirePath (acc_handle_path)
accTerminal (acc_handle_terminal, acc_next_cell_load, acc_next_driver, and
    acc_next_load)
accPathTerminal (acc_next_input and acc_next_output)
accTchkTerminal (acc_handle_tchkarg1 and acc_handle_tchkarg2)
accPartSelect (acc_handle_conn, acc_handle_pathin, and acc_handle_pathout)
accRegBit (acc_handle_by_name, acc_handle_tfarg, and acc_handle_itfarg)
```

If your PLI application uses these types of objects, then it is important to call acc_close to free the memory allocated for these objects when the application is done using them.

## Third party PLI applications

Many third party PLI applications come with instructions on using them with Model*Sim* Verilog. Even without the instructions, it is still likely that you can get it to work with Model*Sim* Verilog as long as the application uses standard PLI routines. The following guidelines are for preparing a Verilog-XL PLI application to work with Model*Sim* Verilog.

Generally, a Verilog-XL PLI application comes with a collection of object files and a veriuser.c file. The veriuser.c file contains the registration information as described above in "Registering PLI applications". To prepare the application for Model*Sim* Verilog, you must compile the veriuser.c file and link it to the object files to create a dynamically loadable object (see "Compiling and linking PLI applications"). For example, if you have a veriuser.c file and a library archive libapp.a file that contains the application's object files, then the following commands should be used to create a dynamically loadable object for the Solaris operating system:

```
% cc -c -I<install_dir>/modeltech/include veriuser.c
% ld -G -o app.sl veriuser.o libapp.a
```

That's all there is to it. The PLI application is ready to be run with Model*Sim* Verilog. All that's left is to specify the resulting object file to the simulator for loading using the Veriuser modesim.ini file entry, the -pli simulator option, or the PLIOBS environment variable (see "Registering PLI applications").

**Note:** On the HP700 platform, the object files must be compiled as position-independent code by using the +z compiler option. Since, the object files supplied for Verilog-XL may be compiled for static linking, you may not be able to use the object files to create a dynamically loadable object for Model*Sim* Verilog. In this case, you must get the third party application vendor to supply the object files compiled as position-independent code.

## Support for VHDL objects

The PLI ACC routines also provide limited support for VHDL objects in a mixed VHDL/Verilog design. The following table lists the VHDL objects for which handles may be obtained and their type and fulltype constants:

| Type | Fulltype | Description |
|---|---|---|
| accArchitecture | accArchitecture | instantiation of an architecture |
| accArchitecture | accEntityVitalLevel0 | instantiation of an architecture whose entity is marked with the attribute VITAL_Level0 |
| accArchitecture | accArchVitalLevel0 | instantiation of an architecture which is marked with the attribute VITAL_Level0 |
| accArchitecture | accArchVitalLevel1 | instantiation of an architecture which is marked with the attribute VITAL_Level1 |
| accArchitecture | accForeignArch | instantiation of an architecture which is marked with the attribute FOREIGN and which does not contain any VHDL statements or objects other than ports and generics |
| accArchitecture | accForeignArchMixed | instantiation of an architecture which is marked with the attribute FOREIGN and which contains some VHDL statements or objects besides ports and generics |

| Type | Fulltype | Description |
|---|---|---|
| accBlock | accBlock | block statement |
| accForLoop | accForLoop | for loop statement |
| accForeign | accShadow | foreign scope created by mti_CreateRegion( ) |
| accGenerate | accGenerate | generate statement |
| accPackage | accPackage | package declaration |
| accSignal | accSignal | signal declaration |

The type and fulltype constants for VHDL objects are defined in the *acc_vhdl.h* include file. All of these objects (except signals) are scope objects that define levels of hierarchy in the Structure window. Currently, the PLI ACC interface has no provision for obtaining handles to generics, types, constants, attributes, subprograms, and processes.

## IEEE Std 1364 ACC routines

Model*Sim* Verilog supports the following ACC routines, described in detail in the IEEE Std 1364

| | | |
|---|---|---|
| acc_append_delays | acc_append_pulsere | acc_close |
| acc_collect | acc_compare_handles | acc_configure |
| acc_count | acc_fetch_argc | acc_fetch_argv |
| acc_fetch_attribute | acc_fetch_attribute_int | acc_fetch_attribute_str |
| acc_fetch_defname | acc_fetch_delay_mode | acc_fetch_delays |
| acc_fetch_direction | acc_fetch_edge | acc_fetch_fullname |
| acc_fetch_fulltype | acc_fetch_index | acc_fetch_location |
| acc_fetch_name | acc_fetch_paramtype | acc_fetch_paramval |
| acc_fetch_polarity | acc_fetch_precision | acc_fetch_pulsere |
| acc_fetch_range | acc_fetch_size | acc_fetch_tfarg |

| | | |
|---|---|---|
| acc_fetch_itfarg | acc_fetch_tfarg_int | acc_fetch_itfarg_int |
| acc_fetch_tfarg_str | acc_fetch_itfarg_str | acc_fetch_timescale_info |
| acc_fetch_type | acc_fetch_type_str | acc_fetch_value |
| acc_free | acc_handle_by_name | acc_handle_calling_mod_m |
| acc_handle_condition | acc_handle_conn | acc_handle_hiconn |
| acc_handle_interactive_scope | acc_handle_loconn | acc_handle_modpath |
| acc_handle_notifier | acc_handle_object | acc_handle_parent |
| acc_handle_path | acc_handle_pathin | acc_handle_pathout |
| acc_handle_port | acc_handle_scope | acc_handle_simulated_net |
| acc_handle_tchk | acc_handle_tchkarg1 | acc_handle_tchkarg2 |
| acc_handle_terminal | acc_handle_tfarg | acc_handle_itfarg |
| acc_handle_tfinst | acc_initialize | acc_next |
| acc_next_bit | acc_next_cell | acc_next_cell_load |
| acc_next_child | acc_next_driver | acc_next_hiconn |
| acc_next_input | acc_next_load | acc_next_loconn |
| acc_next_modpath | acc_next_net | acc_next_output |
| acc_next_parameter | acc_next_port | acc_next_portout |
| acc_next_primitive | acc_next_scope | acc_next_specparam |
| acc_next_tchk | acc_next_terminal | acc_next_topmod |
| acc_object_in_typelist | acc_object_of_type | acc_product_type |
| acc_product_version | acc_release_object | acc_replace_delays |
| acc_replace_pulsere | acc_reset_buffer | acc_set_interactive_scope |
| acc_set_pulsere | acc_set_scope | acc_set_value |
| acc_vcl_add | acc_vcl_delete | acc_version |

## IEEE Std 1364 TF routines

Model*Sim* Verilog supports the following TF routines, described in detail in the IEEE Std 1364.

| | | |
|---|---|---|
| io_mcdprintf | io_printf | mc_scan_plusargs |
| tf_add_long | tf_asynchoff | tf_iasynchoff |
| tf_asynchon | tf_iasynchon | tf_clearalldelays |
| tf_iclearalldelays | tf_compare_long | tf_copypvc_flag |
| tf_icopypvc_flag | tf_divide_long | tf_dofinish |
| tf_dostop | tf_error | tf_evaluatep |
| tf_ievaluatep | tf_exprinfo | tf_iexprinfo |
| tf_getcstringp | tf_igetcstringp | tf_getinstance |
| tf_getlongp | tf_igetlongp | tf_getlongtime |
| tf_igetlongtime | tf_getnextlongtime | tf_getp |
| tf_igetp | tf_getpchange | tf_igetpchange |
| tf_getrealp | tf_igetrealp | tf_getrealtime |
| tf_igetrealtime | tf_gettime | tf_igettime |
| tf_gettimeprecision | tf_igettimeprecision | tf_gettimeunit |
| tf_igettimeunit | tf_getworkarea | tf_igetworkarea |
| tf_long_to_real | tf_longtime_tostr | tf_message |
| tf_mipname | tf_imipname | tf_movepvc_flag |
| tf_imovepvc_flag | tf_multiply_long | tf_nodeinfo |
| tf_inodeinfo | tf_nump | tf_inump |
| tf_propagatep | tf_ipropagatep | tf_putlongp |
| tf_iputlongp | tf_putp | tf_iputp |
| tf_putrealp | tf_iputrealp | tf_read_restart |

| | | |
|---|---|---|
| tf_real_to_long | tf_rosynchronize | tf_irosynchronize |
| tf_scale_longdelay | tf_scale_realdelay | tf_setdelay |
| tf_isetdelay | tf_setlongdelay | tf_isetlongdelay |
| tf_setrealdelay | tf_isetrealdelay | tf_setworkarea |
| tf_isetworkarea | tf_sizep | tf_isizep |
| tf_spname | tf_ispname | tf_strdelputp |
| tf_istrdelputp | tf_strgetp | tf_istrgetp |
| tf_strgettime | tf_strlongdelputp | tf_istrlongdelputp |
| tf_strrealdelputp | tf_istrrealdelputp | tf_subtract_long |
| tf_synchronize | tf_isynchronize | tf_testpvc_flag |
| tf_itestpvc_flag | tf_text | tf_typep |
| tf_itypep | tf_unscale_longdelay | tf_unscale_realdelay |
| tf_warning | tf_write_save | |

## Verilog-XL compatible routines

The following PLI routines are not define in the IEE Std 1364, but Model*Sim* Verilog provides them for compatibility with Verilog-XL.

```
char *acc_decompile_exp(handle condition)
```

This routine provides similar functionality to the Verilog-XL **acc_decompile_expr** routine. The condition argument must be a handle obtained from the acc_handle_condition routine. The value returned by **acc_decompile_exp** is the string representation of the condition expression.

```
char *tf_dumpfilename(void)
```

This routine returns the name of the VCD file.

```
void tf_dumpflush(void)
```

A call to this routine flushes the VCD file buffer (same effect as calling **$dumpflush** in the Verilog code).

```
int tf_getlongsimtime(int *aof_hightime)
```

This routine gets the current simulation time as a 64-bit integer. The low-order bits are returned by the routine, while the high-order bits are stored in the aof_hightime argument.

## PLI tracing

The foreign interface tracing feature is available for tracing user foreign language calls made to the MTI Verilog PLI. Foreign interface tracing creates two kinds of traces: a human-readable log of what functions were called, the value of the arguments, and the results returned; and a set of C-language files to replay what the foreign interface side did.

### The purpose of tracing files

The purpose of the logfile is to aid you in debugging PLI code. The primary purpose of the replay facility is to send the replay file to MTI support for debugging co-simulation problems, or debugging PLI problems for which it is impractical to send thePLI code. MTI still would need the customer to send the VHDL/Verilog part of the design to actually execute a replay, but many problems can be resolved with the trace only.

### Invoking a trace

To invoke the trace, call **vsim** (CR-148) with the **-trace_foreign** option:

#### Syntax

```
vsim
  -trace_foreign <action> [-tag <name>]
```

### Arguments

`<action>`
Specifies one of the following actions:

| Value | Action | Result |
|---|---|---|
| 1 | create log only | writes a local file called "mti_trace_<tag>" |
| 2 | create replay only | writes local files called "mti_data_<tag>.c", "mti_init_<tag>.c", "mti_replay_<tag>.c" and "mti_top_<tag>.c" |
| 3 | create both log and replay | |

`-tag <name>`
Used to give distinct file names for multiple traces. Optional.

### Examples

```
vsim -trace_foreign 1 mydesign
```
Creates a logfile.

```
vsim -trace_foreign 3 mydesign
```
Creates both a logfile and a set of replay files.

```
vsim -trace_foreign 1 -tag 2 mydesign
```
Creates a logfile with a tag of "2".

The tracing operations will provide tracing during all user foreign code-calls, includingPLI user tasks and functions (calltf, checktf, sizetf and misctf routines), and Verilog VCL call-backs.

# 6 - Multiple logfiles, datasets and virtuals

## Chapter contents


Multiple logfiles and datasets . . . . . 95
Opening and viewing datasets . . . . . 96
Using datasets with ModelSim commands . . . . . 98
Restricting the dataset prefix display . . . . . 99

Virtual Objects (User-defined buses, and more) . . . . . 100
Virtual signals . . . . . 101
Virtual functions . . . . . 102
Virtual regions . . . . . 102
Virtual types . . . . . 103

Logfile and virtual commands reference table . . . . . 103


A Model*Sim* simulation can be logged to a WLF file (formerly a WAV file) for future viewing or comparison to a current simulation. By default the logfile is named *vsim.wlf*.

With Model*Sim* release 5.3, you can open more than one logfile for simultaneous viewing in a single Wave window. You can also create virtual signals that are simple logical combinations of, or logical functions of, signals from different logfiles. This capability provides the basic mechanism for comparing simulations.

## Multiple logfiles and datasets

When running a simulation, the logfile from a previous simulation may be opened in view mode and used as a reference. (View mode allows you to view, but not run, a previous simulation.)

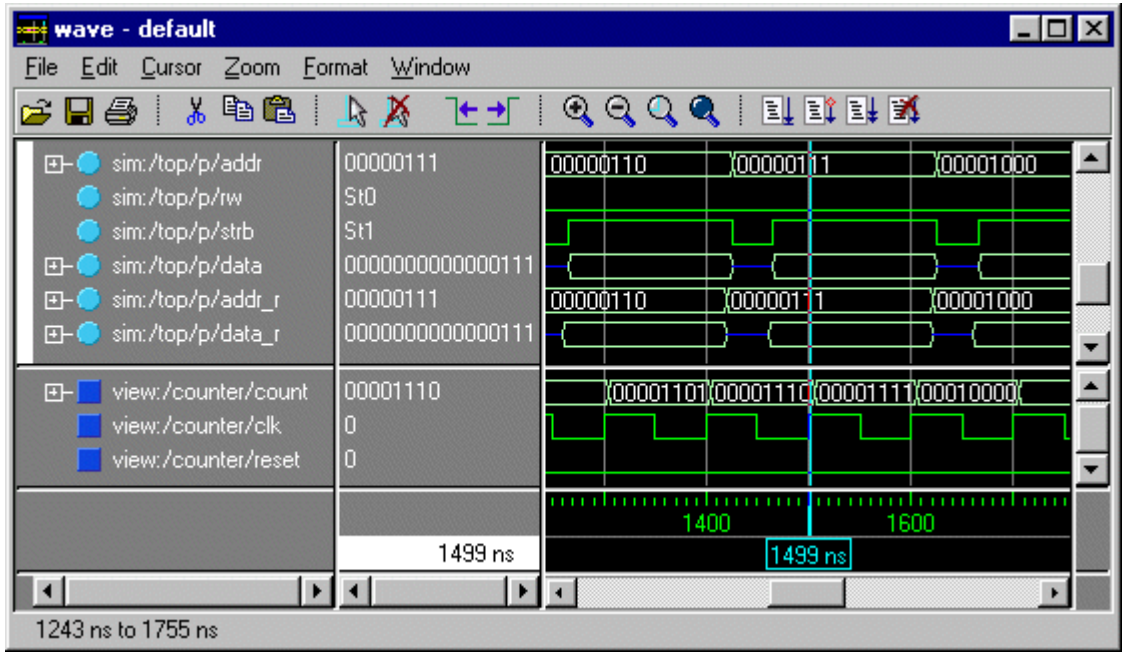
If a logfile is viewed during an active simulation, a unique dataset name identifies each simulation. Two datasets are in view in the Wave window below; the current simulation is indicated by the "sim" prefix and a previous simulation is indicated by the "view" prefix. The default dataset names are:

sim: – for the current active simulation

view: – for the first view-mode file opened

view<n>: – for the (n+1)-th view-mode file opened

Default dataset names may be changed in the *pref.tcl* file.

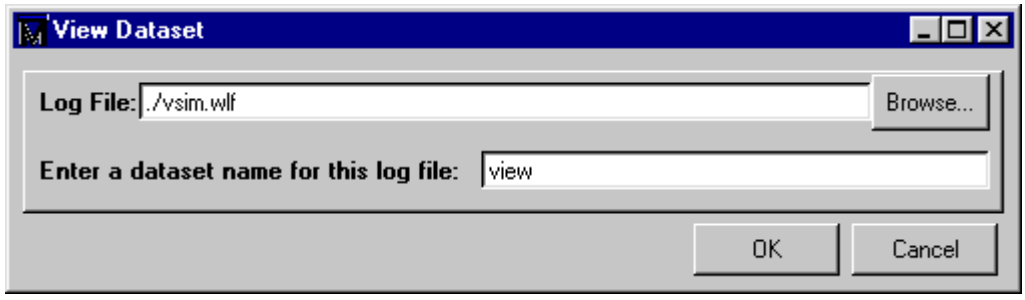



In the illustration above, the Wave window is split into two panes. The top pane shows the dataset of the current simulation. (The default dataset prefix is "sim".) The bottom pane shows a dataset in the view mode only. (The default dataset prefix is "view".) The sim-mode dataset is located in the active pane, as indicated by the white bar in the left margin.

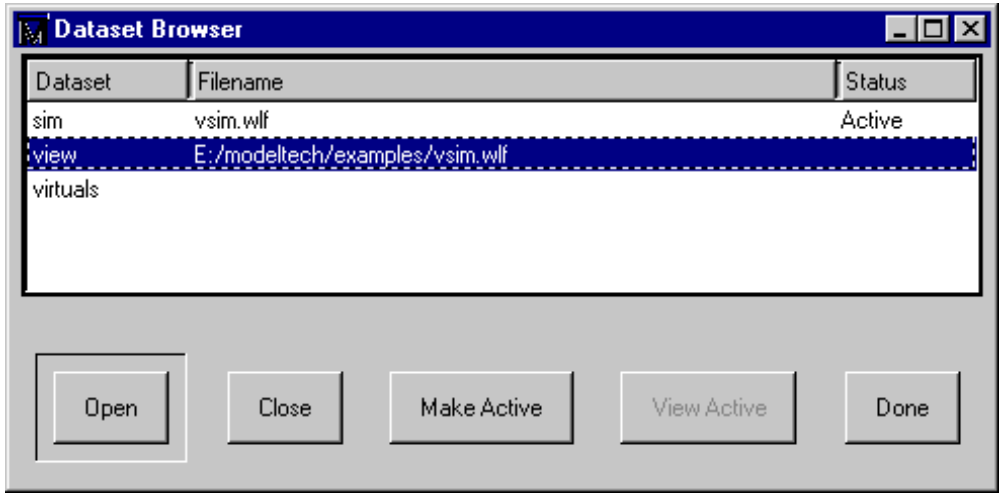
New panes are created with the **Wave > File > New Window Pane** menu selection. When multiple window panes exist, the **add wave** command (CR-23) adds signals to the active pane by default. Click in a pane to make it active.

## Opening and viewing datasets

Logfiles may be opened using either the **Main > File > Open > Open Dataset** or **Wave > Open Dataset** menu selection. Opening datasets in this manner returns the View Dataset dialog box, allowing you to specify the dataset name for the logfile.




Once you have specified the logfile and dataset name, the dataset is ready for viewing. To view the dataset, use the **Main > View > Dataset** menu selection to open the Dataset Browser




Select **Open** to browse for a dataset (this opens the View Dataset dialog box as well). Once the dataset is open, you can select it and choose **Make Active** to prepare it for viewing. Add signals to the Wave window with **add wave** command (CR-23).

**Make Active** makes the selected dataset the "active" or default dataset. Default dataset means that if you type a region path as part of a command and omit the dataset prefix, the default dataset will be assumed. It is equivalent to typing: env <dataset>: at the VSIM prompt.

### Virtuals

Model*Sim* supports an additional, subterranean, dataset named "virtuals", which contains references to user-defined buses and other virtual objects (see below). Normally, you will not need to directly reference the virtuals dataset.

## Using datasets with Model*Sim* commands

Multiple logfiles may be opened when the simulator is invoked by specifying more than one **vsim -view <filename>** option. Dataset prefixes for logfiles opened in this manner will be "view", "view2", "view3", etc.

A dataset name may also be specified as an optional qualifier to the **vsim -view** switch on the command line using the following syntax:

```
-view <dataset>=<filename>
```

For example: **vsim -view foo=vsim.wlf**

Design regions and signal names can be fully specified over multiple logfiles by using the dataset name as a prefix in the path. For example:

```
sim:/top/alu/out
view:/top/alu/out
golden:top.alu.out
```

Dataset prefixes are not required unless more than one dataset is open and you want to refer to something outside the default dataset. When more than one dataset is open, Model*Sim* will automatically prefix names in the Wave and List window with the dataset name. This may be changed using the **Props > Display Properties** dialog in those windows.

Model*Sim* designates one of the datasets to be the "current" dataset, and refers all names without dataset prefixes to that dataset. The current dataset is displayed in the context path at the bottom of the Main window. It can be displayed using the **environment** command (CR-50) with no arguments, or with the **View > Datasets** menu selection.

The Structure and Signals windows each have a current dataset to which they are sensitive. Being sensitive to a dataset means that the window will update when the content of that dataset changes. The dataset to which these windows are sensitive is set using the **File > Environment** menu selection in the respective window.

Additionally, a Structure or Signals window may be created sensitive to a specified dataset using the **-env <dataset>** switch to the **view** command. For example,

```
view -new signals -env view1
```

will create a new Signals window sensitive to the **view1** dataset.

Model*Sim* remembers a "current context" within each open dataset. You can toggle between the current context of each dataset using the **environment** command specifying the dataset without a path. For example:

```
env foo:
```

will set the current dataset to **foo** and the current context to the context last specified for **foo**.

The current context of the current dataset (usually referred to as just "current context") is used for finding objects specified without a path.

## Restricting the dataset prefix display

The default for dataset prefix viewing is set with a variable in *pref.tcl*, **PrefMain(DisplayDatasetPrefix)**. Setting the variable to 1 will display the prefix, setting it to 0 will not. It is set to 1 by default. You can use the Tcl **set** command to change the variable value from the Model*Sim* command line:

```
set PrefMain(DisplayDatasetPrefix) 0
```

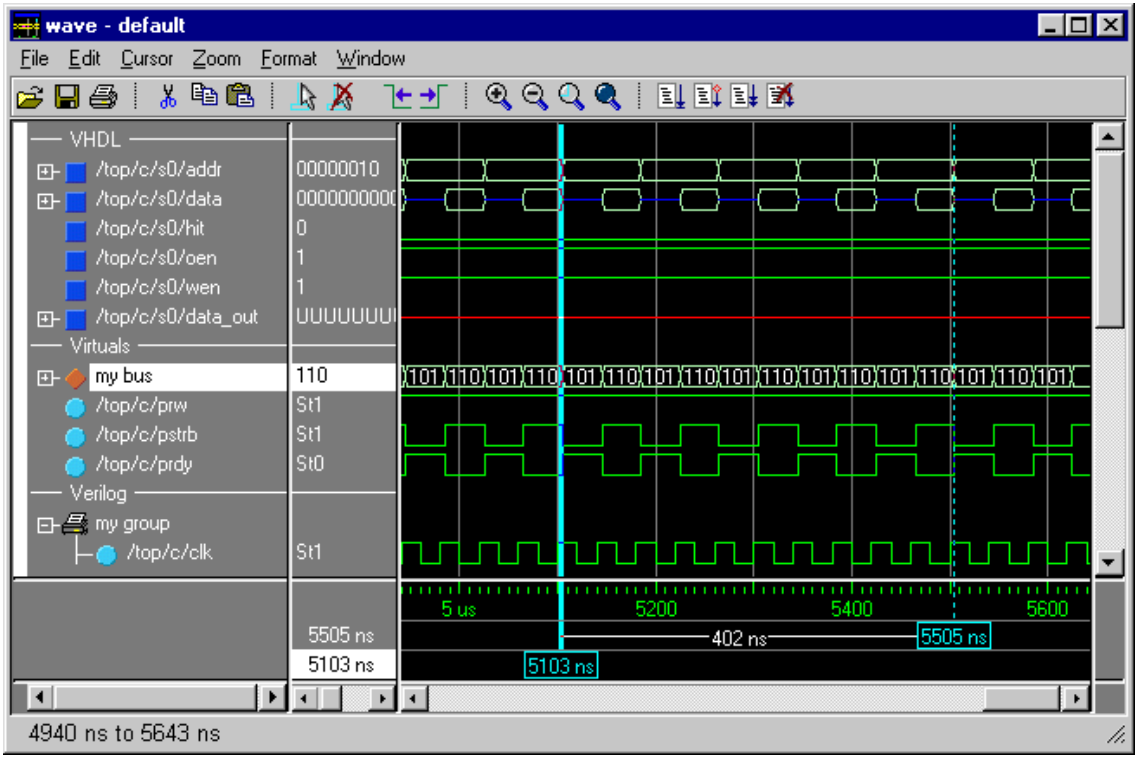
Additionally, you can restrict display of the dataset prefix if you use the **environment -nodataset** command to view a dataset. To display the prefix use the **environment** command (CR-50) with the **-dataset** option (you won't need to specify this option if the variable noted above is set to 1). The **environment** command line switches override the *pref.tcl* variable.

# Virtual Objects (User-defined buses, and more)

Virtual objects are signal-like or region-like objects created in the GUI that do not exist in the Model*Sim* simulation kernel. Model*Sim* release 5.3 supports the following kinds of virtual objects:

- Virtual signals (6-101)
- Virtual functions (6-102)
- Virtual regions (6-102)
- Virtual types (6-103)

Virtual objects are indicated by an orange diamond as illustrated by *my bus* below:



## Virtual signals

Virtual signals are aliases for combinations or subelements of signals written to the logfile by the simulation kernel. They may be displayed in the Signals, List or Wave window, accessed by the **examine** command, and set using the **force** command. Virtual signals may be created by menu selections in the Signals, Wave or List windows, or with the **virtual signal** command described below. Virtual signals can also be dragged and dropped from the Signals window to the Wave or List window.

Virtual signals are automatically attached to the design region in the hierarchy that corresponds to the nearest common ancestor of all the elements of the virtual signal. The **virtual signal** command has an **-install <region>** option to specify where the virtual should be installed. This may be used to install the virtual signal in a user-defined region in order to reconstruct the original RTL hierarchy when simulating and driving a post-synthesis, gate-level implementation.

A virtual signal can be used to reconstruct RTL-level design buses that were broken down during synthesis. The **virtual hide** command can be used to hide the display of the broken-down bits if you don't want them cluttering up the Signals window (see below).

If the virtual signal has elements from more than one logfile, it will be automatically installed in the virtual region "virtuals:/Signals."

Virtual signals are not hierarchical – if two virtual signals are concatenated to become a third virtual signal, the resulting virtual signal will be a concatenation of all the subelements of the first two virtual signals.

The definitions of virtuals can be saved to a macro file using the **virtual save** command (see below). By default, when quitting, Model*Sim* will append any newly-created virtuals (that have not been saved) to the *virtuals.do* file in the local directory.

If you have virtual signals displayed in the Wave or List window when you save the Wave or List format, you will need to execute the *virtuals.do* file (or some other equivalent) to restore the virtual signal definitions before you re-load the Wave or List format during a later run.

### Implicit and explicit virtuals

There is one exception: "implicit virtuals" are automatically saved with the Wave or List format. An implicit virtual is a virtual signal that was automatically created by Model*Sim* without your knowledge and without you providing a name for it. An example would be if you expand a bus in the Wave window, then drag one bit out of the bus to display it separately. That action creates a one-bit virtual signal

whose definition is stored in a special location, and is not visible in the Signals window or to the normal virtual commands.

All other virtual signals are considered "explicit virtuals".

## Virtual functions

Virtual functions behave in the GUI like signals but are not aliases of combinations or elements of signals logged by the kernel. They consist of logical operations on logged signals and may be dependent on simulation time. They may be displayed in the Signals, Wave or List windows, accessed by the **examine** command, but cannot be set by the **force** command.

Examples of virtual functions include the following:

- a function defined as the inverse of a given signal
- a function defined as the exclusive-OR of two signals
- a function defined as a repetitive clock
- a virtual function defined as "the rising edge of CLK delayed by 1.34 ns"

Virtual functions can also be used to convert signal types and map signal values.

The result type of a virtual signal can be any of the types supported in the GUI expression syntax: integer, real, boolean, std_logic, std_logic_vector, and arrays and records of these types. Verilog types are converted to VHDL 9-state std_logic equivalents and Verilog net strengths are ignored.

Virtual functions can be created using the **virtual function** command (see below).

Virtual functions are also implicitly created by Model*Sim* when referencing bit-selects or part-selects of Verilog registers in the GUI, or when expanding Verilog registers in the Signals, Wave or List windows. This is necessary because referencing Verilog register elements requires an intermediate step of shifting and masking of the Verilog "vreg" data structure.

## Virtual regions

User-defined design hierarchy regions may be defined and attached to any existing design region or to the virtuals context tree. They may be used to reconstruct the RTL hierarchy in a gate-level design, and used to locate virtual signals. Thus, virtual signals and virtual regions may be used in a gate-level design to allow the RTL test bench with the gate-level design.

Virtual regions are created and attached using the **virtual region** command (see below).

### Virtual types

User-defined enumerated types may be defined in order to display signal bit sequences as meaningful alphanumeric names. The virtual type is then used in a type conversion expression to convert a signal to values of the new type. When the converted signal is displayed in any of the windows, the value will be displayed as the enumeration string corresponding to the value of the original signal.

## Logfile and virtual commands reference table

The table below provides a brief description of the actions associated with logfile and virtual commands. For complete details about command syntax, arguments and usage, refer to the *ModelSim Command Reference*.

| Command name | Action |
|---|---|
| searchlog (CR-88) | searches one or more of the currently open logfiles for a specified condition |
| virtual count (CR-117) | counts the number of explicitly declared virtuals that have not been saved and that were not read in using a macro file |
| virtual define (CR-118) | prints the definition of the virtual signal or function in the form of a command that can be used to re-create the object |
| virtual delete (CR-119) | removes the matching virtuals |
| virtual describe (CR-120) | prints a complete description of the data type of one or more virtual signals |
| virtual expand (CR-121) | produces a list of all the non-virtual objects contained in the virtual signal(s) |
| virtual function (CR-122) | creates a new signal that consists of logical operations on existing signals and simulation time |
| virtual hide (CR-126) | sets a flag in the specified real or virtual signals so that the signals do not appear in the Signals window |
| virtual log (CR-127) | causes the sim-mode dependent signals of the specified virtual signals to be logged by the kernel |
| virtual nohide (CR-129) | resets the flag set by a virtual hide command |

| Command name | Action |
|---|---|
| virtual nolog (CR-130) | causes the specified virtual signals to be un-logged by the kernel |
| virtual region (CR-132) | creates a new user-defined design hierarchy region |
| virtual save (CR-133) | saves the definitions of virtuals to a file |
| virtual show (CR-134) | lists the full path names of all the virtuals explicitly defined |
| virtual signal (CR-135) | creates a new signal that consists of concatenations of signals and subelements |
| virtual type (CR-138) | creates a new enumerated type |

# 7 - ModelSim XE Graphic Interface

## Chapter contents


Window overview . . . . . 106

Window features . . . . . 107
- Main window . . . . . 113
- Dataflow window . . . . . 124
- List window . . . . . 128
- Process window . . . . . 142
- Signals window . . . . . 144
- Source window . . . . . 152
- Structure window . . . . . 158
- Variables window . . . . . 161
- Wave window . . . . . 164

Compiling with the graphic interface . . . . . 195

Setting default compile options . . . . . 196

Simulating with the graphic interface . . . . . 202

Setting default simulation options . . . . . 211


The example graphics in this chapter illustrate Model*Sim*'s graphic interface within a Windows 98/NT environment.

## Window overview

The Model*Sim* simulation and debugging environment consists of nine windows. A brief description of each window follows:

- Main window (7-113)
  The main window from which all subsequent VSIM windows are available.
- Dataflow window (7-124)
  Lets you trace signals and nets through your design by showing related processes.
- List window (7-128)
  Shows the simulation values of selected VHDL signals, and Verilog nets and register variables in tabular format.
- Process window (7-142)
  Displays a list of processes that are scheduled to run during the current simulation cycle.
- Signals window (7-144)
  Shows the names and current values of VHDL signals, and Verilog nets and register variables in the region currently selected in the Structure window.
- Source window (7-152)
  Displays the HDL source code for the design. (Your source code can remain hidden if you wish, see "Source code security and -nodebug" (C-292).)
- Structure window (7-158)
  Displays the hierarchy of structural elements such as VHDL component instances, packages, blocks, generate statements, and Verilog model instances, named blocks, tasks and functions.
- Variables window (7-161)
  Displays VHDL constants, generics, variables, and Verilog register variables in the current process and their current values.
- Wave window (7-164)
  Displays waveforms, and current values for the VHDL signals, and Verilog nets and register variables you have selected. Current and past simulations can be compared side-by-side in one Wave window.

## Window features

ModelSim's graphic interface provides many features that add to its usability; features common to many of the windows are described below.

| Feature | Feature applies to these windows |
|---|---|
| Quick access toolbars (7-108) | Main, Source, and Wave |
| Drag and Drop (7-108) | Dataflow, List, Signals, Source, Structure, Variables, and Wave windows |
| Command history (7-109) | Main window command line |
| Automatic window updating (7-109) | Dataflow, Process, Signals, and Structure |
| Finding names, and locating cursors (7-110) | various windows |
| Sorting HDL items (7-110) | Process, Signals, Source, Structure, Variables and Wave windows |
| Menu tear off (7-110) | all windows |
| Combine signals into a user-defined bus (7-110) | List and Wave windows |
| Tree window hierarchical view (7-111) | Structure, Signals, Variables, and Wave windows |

- Press the <ESC> key to cancel any dialog box.
- Cut/Copy/Paste/Delete into any entry box by clicking the right mouse button in the entry box.

- Standard cut/copy/paste shortcut keystrokes – ^X/^C/^V – will work in all entry boxes.
- When the focus changes to an entry box, the contents of that box are selected (highlighted). This allows you to replace the current contents of the entry box with new contents with a simple paste command, without having to delete the old value.
- Dialog boxes will appear on top of their parent window (instead of the upper left corner of the screen)

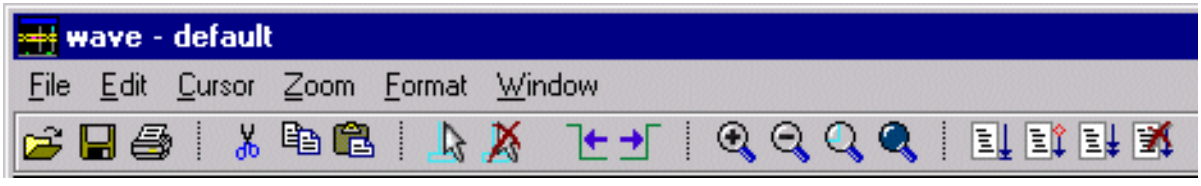
- The transcript window now includes an edit popup menu activated via with the right mouse button.
- The middle mouse button will allow you to paste the following into the transcript window:
  - text currently selected in the transcript window,
  - a current primary X-Windows selection (may be from another application), or
  - contents of the clipboard.

Transcript edit popup

**Note:** Selecting text in the transcript window makes it the current primary X-Windows selection. This way you can copy transcript window selections to other X-Windows windows (xterm, emacs, etc.).

- The **Edit > Paste** operation in the transcript window will ONLY paste from the clipboard.
- All menus highlight their accelerator keys.

## Quick access toolbars




Buttons on the Main, Source, and Wave windows provide access to commonly used commands and functions. See, "The Main window tool bar" (7-119), "The Source window tool bar" (7-155), and "Wave window tool bar" (7-172).

## Drag and Drop

Drag and drop of HDL items is possible between the following windows. Using the left mouse button, click and release to select an item, then click and hold to drag it.

- **Drag items from these windows:**
  Dataflow, List, Signals, Source, Structure, Variables, and Wave windows

- **Drop items into these windows:**
  List and Wave windows

**Note:** Drag and drop works to rearrange items *within* the List and Wave windows as well.

## Command history

Avoid entering long commands twice; use the down and up keyboard arrows to move through the command history for the current simulation.

## Automatic window updating

Selecting an item in the following windows automatically updates other related Model*Sim* windows as indicated below:

| Select an item in this window | To update these windows |
|---|---|
| Dataflow window (7-124)<br><br>(with a process selected in the center of the window) | Process window (7-142) |
| | Signals window (7-144) |
| | Source window (7-152) |
| | Structure window (7-158) |
| | Variables window (7-161) |
| Process window (7-142) | Dataflow window (7-124) |
| | Signals window (7-144) |
| | Structure window (7-158) |
| | Variables window (7-161) |
| Signals window (7-144) | Dataflow window (7-124) |
| Structure window (7-158) | Signals window (7-144) |
| | Source window (7-152) |

## Finding names, and locating cursors

- **Find** HDL item names with the **Edit > Find** menu selection in these windows: List, Process, Signals, Source, Structure, Variables, and Wave windows.

You can also:

- **Locate** time markers in the List window with the **Markers > Goto** menu selection.
- **Locate** time cursors in the Wave window with the **Cursor > Goto** menu selection.

## Sorting HDL items

Use the **Edit > Sort** menu selection in the windows below to sort HDL items in ascending, descending or declaration order.

Process, Signals, Source, Structure, Variables and Wave windows

Names such as net_1, net_10, and net_2 will sort numerically in the Signals and Wave windows.

## Menu tear off

All window menus may be "torn off " to create a separate menu window. To tear off, click on the menu, then select the dotted-line button at the top of the menu.

## Combine signals into a user-defined bus

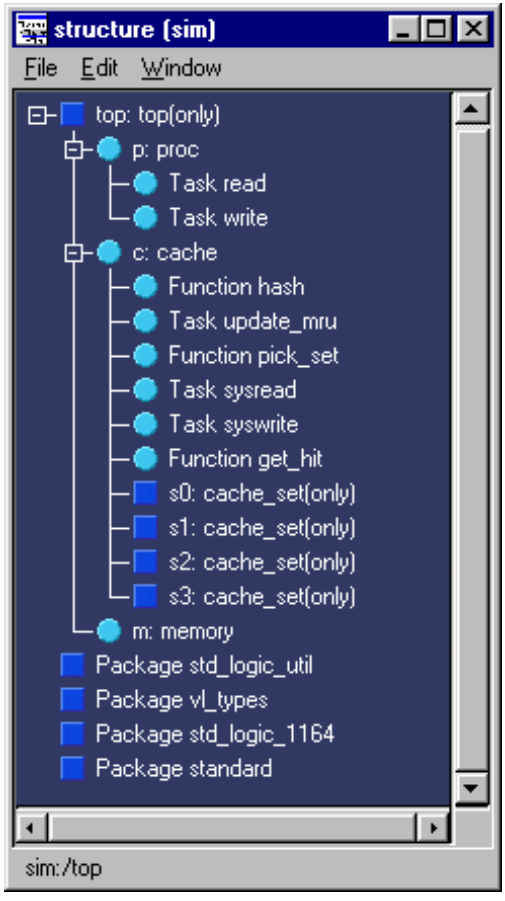
You can collect selected items in the List window (7-128) and Wave window (7-164) displays and combine them into a bus named by you. In the List window, the **Edit > Combine** menu selection allows you to move the selected items to the new bus as long as they are all scalars or arrays of the same base type (records are not yet supported).

In the Wave window (7-164), the **Edit > Combine** menu selection requires all selected items to be either all scalars or all arrays of the same size. The benefit of this added restriction is that the bus can be expanded to show each element as a separate waveform. Using the **flatten** option allows scalars and various array sizes to be mixed, but foregoes display of child waveforms.

The **keep** option in both windows copies the signals rather than moving them.

## Tree window hierarchical view

ModelSim provides a hierarchical, or "tree view" of some aspect of your design in the Structure, Signals, Variables, and Wave windows.




### HDL items you can view

Depending on which window you are viewing, one entry is created for each of the following VHDL and Verilog HDL item within the design:

*VHDL items*

(indicated by a dark blue square icon)
signals, variables, component instantiation, generate statement, block statement, and package

*Verilog items*

(indicated by a lighter blue circle icon)
parameters, registers, nets, module instantiation, named fork, named begin, task, and function

*Virtual items*

(indicated by an orange diamond icon)
virtual signals, buses, and functions, see "Virtual Objects (User-defined buses, and more)" (6-100) for more information

### Viewing the hierarchy

Whenever you see a tree view, as in the Structure window displayed here, you can use the mouse to collapse or expand the hierarchy. Select the symbols as shown below to change the view of the structure.

| Symbol | Description |
|---|---|
| [ + ] | click a plus box to expand the item and view the structure |
| [ - ] | click a minus box to hide a hierarchy that has been expanded |

### Finding items within tree windows

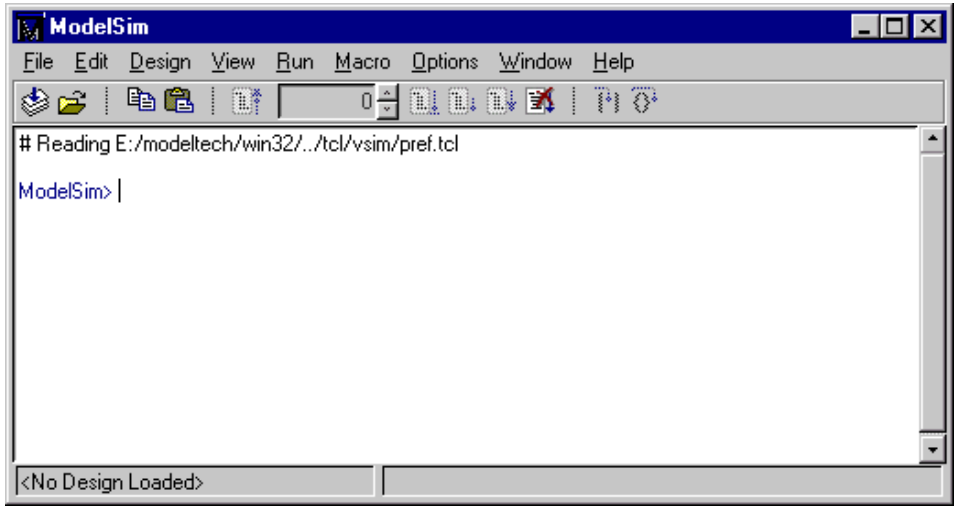
You can open the find dialog box within all windows (except the Main, and Source windows) by using **<control-f>**.

Options within the Find dialog box allow you to search unique text-string fields within the specific window. See also,

- "Finding items by name in the List window" (7-139),
- "Finding HDL items in the Signals window" (7-150), and
- "Finding items by name or value in the Wave window" (7-183).

## Main window

The Main window is pictured below as it appears when VSIM is first invoked. Note that your operating system graphic interface provides the window-management frame only; Model*Sim* handles all internal-window features including menus, buttons, and scroll bars.



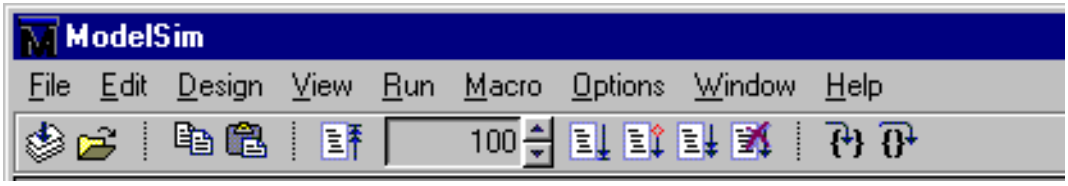

The menu bar at the top of the window provides access to a wide variety of simulation commands and Model*Sim* preferences. The status bar at the bottom of the window gives you information about the data in the active Model*Sim* window. The tool bar provides buttons for quick access to the many common commands.

When a simulation is running, the Main window displays a VSIM prompt, allowing you to enter command-line commands from within the graphic interface. Messages output by VSIM during simulation are also displayed in this window. You can scroll backward and forward through the current work history by using the vertical scrollbar. You can also copy and paste using the mouse within the window, see "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121).

The Main window menu bar, tool bar, and status bar are detailed below.

## The Main window menu bar

The menu bar at the top of the Main window lets you access many Model*Sim* commands and features. The menus are listed below with brief descriptions of the command's use.




### File menu

| | |
|---|---|
| New | provides the following four options:<br>New Folder – create a new folder in the current directory<br>New Source – create a project source file<br>Import Source – import a project source file<br>New Project – create a new project from scratch or copy an existing project |
| Open | provides the following three options:<br>Open Source – opens the selected project source file<br>Open Project – opens the selected .mpf project file<br>Open Dataset – view new design simulation logfile (.wlf); enter short name for this design view |
| Delete | provides the following two options:<br>Delete Project – delete the selected .mpf project file<br>Delete Source – delete the selected project source file |
| Change Directory | change to a different working directory |
| Save Transcript | save the current contents of the transcript window to the file indicated with a "Save Transcript as" selection (this selection is not initially available because the transcript is written to the *transcript* file by default), see "Saving the Main window transcript file" (7-118) |
| Save Transcript as... | save the current contents of the transcript window to a file |
| Clear Transcript | clear the Main window transcript display |

| | |
|---|---|
| Options<br>(all options are set for the current session only) | Transcript File: sets a transcript file to save for this session only<br>Command History: file for saving command history only, no comments<br>Save File: sets filename for Save Main, and Save Main as<br>Saved Lines: limits the number of lines saved in the transcript (default is all)<br>Line Prefix: specify the comment prefix for the transcript<br>Update Rate: specify the update frequency for the Main status bar<br>Model*Sim* Prompt: change the title of the Model*Sim* prompt<br>VSIM Prompt: change the title of the VSIM prompt<br>Paused Prompt: change the title of the Paused prompt |
| <path list> | **Windows only** - a list of the most recent working directory changes |
| Quit | quit Model*Sim* |

### Edit menu

| | |
|---|---|
| Copy | copy the selected text |
| Paste | paste the previously cut or copied item to the left of the currently selected item |
| Select All | select all text in the Main window transcript |
| Unselect All | deselect all text in the Main window transcript |
| Find | search the transcript forward or backward for the specified text string |

### Design menu

| | |
|---|---|
| Browse Libraries | browse all libraries within the scope of the design |
| Create a New Library | create a new library or map a library to a new name |
| View Library Contents | view or delete the contents of a library |
| Compile | compiles HDL source files into the current project's work library |
| Compile Project | recompile all of the previously compiled files in the current project |
| Load New Design | initiate simulation by specifying top level design unit in the Design tab; specify HDL specific simulator settings with the VHDL and Verilog tabs; specify setting relating to the annotation of design timing with the SDF tab |
| End Simulation | end the simulation |

### View menu

| | |
|---|---|
| All | open all VSIM windows |
| Source | open and/or view the Source window (7-152) |
| Structure | open and/or view the Structure window (7-158) |
| Variables | open and/or view the Variables window (7-161) |
| Signals | open and/or view the Signals window (7-144) |
| List | open and/or view the List window (7-128) |
| Process | open and/or view the Process window (7-142) |
| Wave | open and/or view the Wave window (7-164) |
| Dataflow | open and/or view the Dataflow window (7-124) |
| Datasets | opens the Dataset Browser for selecting the current Dataset |

### Run menu

| | |
|---|---|
| Run <default> | run simulation for one default run length; change the run length with Options > Simulation, or use the Run Length list on the tool bar |
| Run -All | run simulation until you stop it |
| Continue | continue the simulation |
| Run -Next | run to the next event time |
| Step | single-step the simulator |
| Step-Over | execute without single-stepping through a subprogram call |
| Restart | reloads the design elements and resets the simulation time to zero; only design elements that have changed are reloaded; you specify whether to maintain the following after restart—list and wave window environment, breakpoints, logged signals, and virtual definitions; see also the **restart** command (CR-84) |

### Macro menu

| | |
|---|---|
| Execute Macro | allows you to browse for and execute a DO file (macro) |
| Execute Old PE Macro... | calls and executes old PE 4.7 macro without changing the macro to XE 5.3; backslashes may be selected as pathname delimiters |
| Convert Old PE Macro... | converts old PE 4.7 macro to XE 5.3 macro without changing the file; backslashes may be selected as pathname delimiters |

### Options menu

| | |
|---|---|
| Compile | returns the Compile Options dialog box; options cover both VHDL and Verilog compile options |
| Simulation | returns the Simulation Options dialog box; options include: default radix, default force type, default run length, iteration limit, warning suppression, and break on assertion specification |
| Edit Preferences... | returns the Preferences dialog box; color preferences can be set for window background, text and graphic items (i.e., waves in the Wave window) |
| Edit Project | allows modification of project-wide settings that describe the makeup of the project |
| Save Preferences | save current Model*Sim* settings to a Tcl preference file; saves preferences as Tcl arrays |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |

| | |
|---|---|
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window |

### Help menu

| | |
|---|---|
| About Model*Sim* | display Model*Sim* application information |
| Release Notes | view current release notes with the Model*Sim* **notepad** (CR-70) |
| Enable Welcome | enables Welcome screen for starting a new project or opening an existing project when Model*Sim* is initiated |
| Quick Start Menu | enables Quick Start, which explains the process of setting up a project, compiling project source files, simulating a project and modifying it; also includes three step by step simulation examples |
| Information about Help | view the readme file pertaining to Model*Sim*'s online documentation |
| XE User's Manual and Command Reference | open and read the Model*Sim* documentation in PDF format; PDF files can be read with a free Adode Acrobat reader available through www.adobe.com |
| Tcl Help | open the Tcl command reference (man pages) in Windows help format |
| Tcl Syntax | open and read Tcl syntax details in HTML format |
| Tcl Man Pages | open and read Tcl /Tk 8.0 manual in HTML format |
| Technotes | select a technical note to view from the drop-down list |

## Saving the Main window transcript file

Variable settings determine the filename used for saving the Main window transcript. If either PrefMain(file) in *modelsim.tcl*, or TranscriptFile in *modelsim.ini* file is set, then the transcript output is logged to the specified file. By default the TranscriptFile variable in *modelsim.ini* is set to *transcript*. If either variable is set, the transcript contents are always saved and no explicit saving is necessary.

If you would like to save an additional copy of the transcript with a different filename, you can use the File > Save Main As, or File > Save Main menu items. The initial save must be made with the Save Main As selection, which stores the filename in the Tcl variable PrefMain(saveFile). Subsequent saves can be made

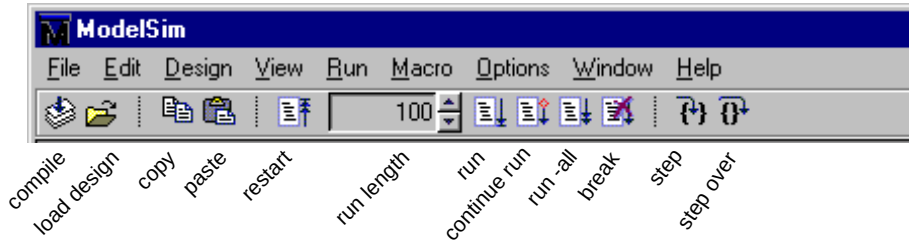
with the Save Main selection. Since no automatic saves are performed for this file, it is written only when a Save... menu selection is made. The file is written to the current working directory and records the contents of the transcript at the time of the save.

### Using the saved transcript as a macro (DO file)

Saved transcript files can be used as macros (DO files). See the **do** command (CR-43) for more information.

## The Main window tool bar

Buttons on the Main window tool bar give you quick access to these Model*Sim* commands and functions.




**Main window tool bar buttons**

| Button | Menu equivalent | Command equivalents |
|---|---|---|
| **Compile** open the Compile HDL Source Files dialog box to select files for compilation | none, however, Options > Compile opens the Compile Options dialog box | **vcom** <arguments>, or **vlog** <arguments><br><br>see: **vcom** (CR-106) or **vlog** (CR-141) |
| **Load Design** open the Load a Design dialog box to initiate simulation | File > Load New Design | **vsim** <arguments><br><br>see: **vsim** (CR-148) |

| Main window tool bar buttons | | |
|---|---|---|
| **Button** | **Menu equivalent** | **Command equivalents** |
| **Copy** copy the selected text within the Main window transcript | Edit > Copy | see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |
| **Paste** paste the copied text to the cursor location | Edit > Paste | see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |
| **Restart** reloads the design elements and resets the simulation time to zero | File > Restart | **restart** <arguments><br><br>see: **restart** (CR-84) |
| **Run Length** specify the run length for the current simulation | none | **run** <specific run length><br><br>see: **run** (CR-86) |
| **Run** run the current simulation for the default time length | Run > Run <default_run_length>… | **run** (no arguments)<br><br>see: **run** (CR-86) |
| **Continue Run** continue the current simulation run | Run > Continue | **run -continue**<br><br>see: **run** (CR-86) |
| **Run -All** run to current simulation forever, or until it hits a breakpoint or specified break event * | Run > Run -All | **run -all**<br><br>see: **run** (CR-86), * see "Assertion settings page" (7-213) |
| **Break** stop the current simulation run | none | none |

| Main window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Command equivalents** |
| | **Step**<br>steps the current simulation to the next HDL statement | Run > Step…. | **step**<br><br>see: **step** (CR-93) |
| | **Step Over**<br>HDL statements are executed but treated as simple statements instead of entered and traced line by line | Run > Step Over…. | **step -over**<br><br>see: **step** (CR-93) |

## The Main window status bar

Fields at the bottom of the Main window provide the following information about the current simulation:

| Field | Description |
|---|---|
| Now | the current simulation time, using the default resolution units specified in "Simulating with the graphic interface" (7-202), or a larger time unit if one can be used without a fractional remainder |
| Delta | the current simulation iteration number |
| <dataset name> | name of the current dataset (item selected in the Structure window (7-158)) |

## Mouse and keyboard shortcuts in the Transcript and Source windows

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file

displayed in the Source window and all Notepad windows (enter the **notepad** command within Model*Sim* to open the Notepad editor).

| Mouse | Result |
|---|---|
| < left-button - click > | move the insertion cursor |
| < left-button - press > + drag | select |
| < shift - left-button - press > | extend selection |
| < left-button - double-click > | select word |
| < left-button - double-click > + drag | select word + word |
| < control - left-button - click > | move insertion cursor without changing the selection |
| < left-button - click > on previous ModelSim or VSIM prompt | copy and paste previous command string to current prompt |

| Keystrokes | Result |
|---|---|
| < left \| right - arrow > | move the insertion cursor |
| < up \| down - arrow > | scroll through command history |
| < control - p > | move insertion cursor to previous line |
| < control - n > | move insertion cursor to next line |
| < control - f > | move insertion cursor forward |
| < control - b > | move insertion cursor backward |
| < backspace > | delete character to the left |
| < control - d >, <Delete> | delete character to the right |
| < control - k > | delete to the end of line |
| < control - a >, <Home> | move insertion cursor to beginning of line |

| Keystrokes | Result |
|---|---|
| < control - e >, <Home> | move insertion cursor to end of line |
| < control - x ><br><br>< control - c ><br><br>< control - v > | cut selection |
| | copy selection |
| | insert clipboard |

The Main window allows insertions or pastes only after the prompt, therefore, you don’t need to set the cursor when copying strings to the command line.

## Dataflow window

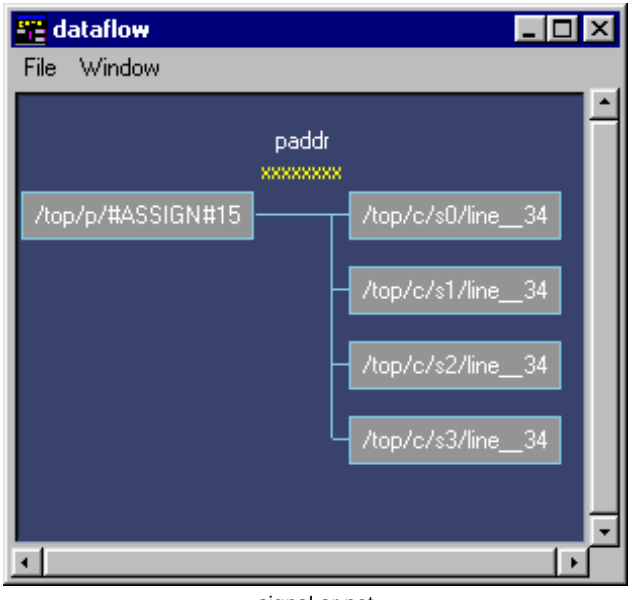
The Dataflow window allows you to trace VHDL signals or Verilog nets through your design. Double-click an item with the left mouse button to move it to the center of the Dataflow display.

**VHDL signals or processes in the Dataflow window:**

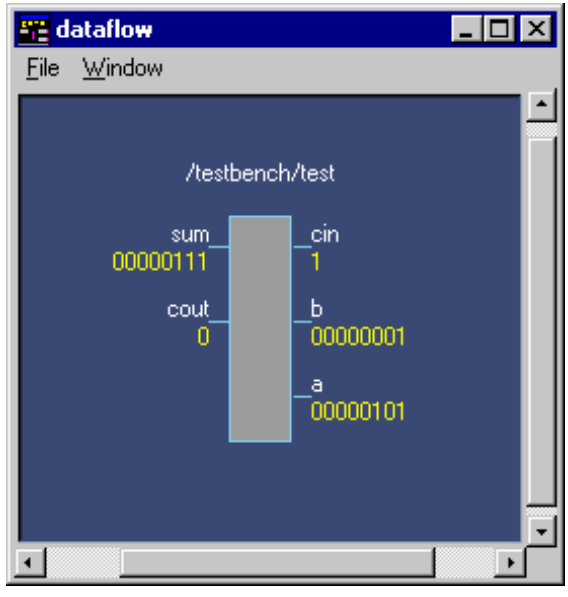
- A signal displays in the center of the window with all the processes that drive the signal on the left, and all the processes that read the signal on the right, or
- a process is displayed with all the signals read by the process shown as inputs on the left of the window, and all the signals driven by the process on the right.

**Verilog nets or processes in the Dataflow window:**

- A net displays in the center of the window with all the processes that drive the net on the left, and all the processes triggered by the net on the right, or
- a process is displayed with all the nets that trigger the process shown as inputs on the left of the window, and all the nets driven by the process on the right.




signal or net




process

## The Dataflow window menu bar

The following menu commands and button options are available from the Dataflow window menu bar.

### File menu

| | |
|---|---|
| Save Postscript | save the current dataflow view as a Postscript file; see "Saving the Dataflow window as a Postscript file" (7-127) |
| Selection | Selection > Follow Selection updates window when the Process window (7-142) or Signals window (7-144) changes; Selection > Fix Selection freezes the view selected from within the Dataflow window |
| Close | close this copy of the Dataflow window |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## Tracing HDL items with the Dataflow window

The Dataflow window is linked with the Signals window (7-144) and the Process window (7-142). To examine a particular process in the Dataflow window, click on the process name in the Process window. To examine a particular HDL item in the Dataflow window, click on the item name in the Signals window.

**with a signal in center** of the Dataflow window, you can:

- click once on a process name in the Dataflow window to make the Source and Variable windows update to show that process,
- click twice on a process name in the Dataflow window to move the process to the center of the Dataflow window

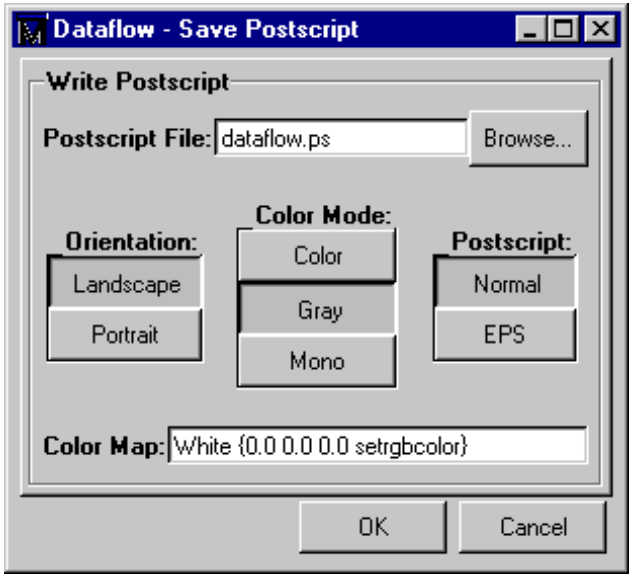
**with a process in center** of the Dataflow window, you can:

- click once on an item name to make the Source and Signals windows update to show that item,
- click twice on an item name to move that item to the center of the Dataflow window.

The Dataflow window will display the current process when you single-step or when VSIM hits a breakpoint.

## Saving the Dataflow window as a Postscript file

Use this Dataflow window menu selection: **File > Save Postscript** to save the current Dataflow view as a Postscript file. Configure the Postscript output with the following dialog box, or use the Preferences dialog box from this Main window selection: **Option > Edit Preferences**.

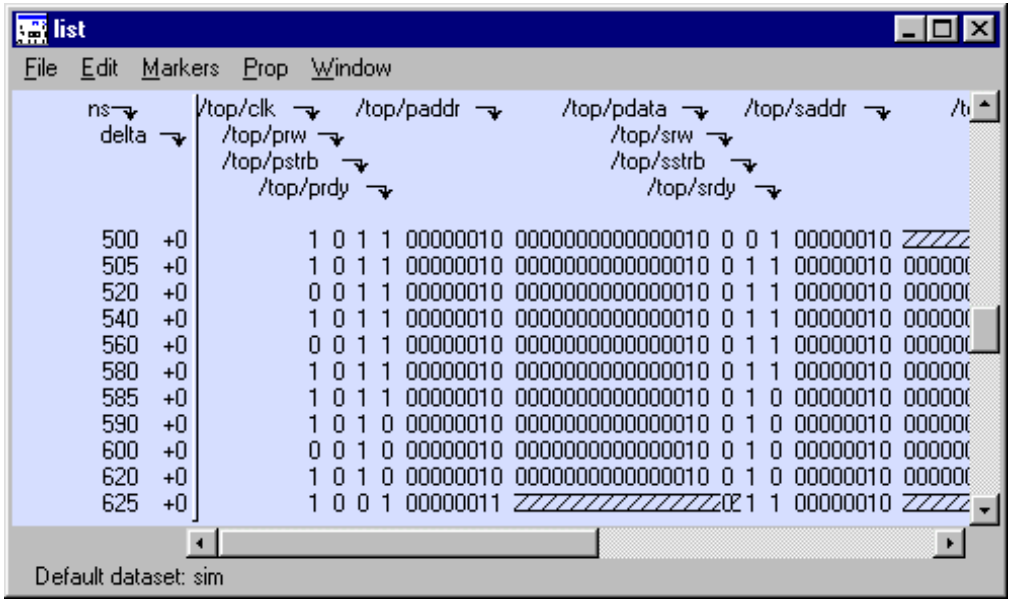



The dialog box has the following options:

- **Postscript File**
specify the name of the file to save, default is *dataflow.ps*
- **Orientation**
specify **Landscape** (horizontal) or **Portrait** (vertical) orientation
- **Color Mode**
specify **Color** (256 colors), **Gray** (gray-scale) or **Mono** color mode
- **Postscript**
specify Normal Postscript or EPS (Encapsulated Postscript) file type
- **Color Map**
specify the color mapping from current Dataflow window colors to Postscript colors

# List window

The List window displays the results of your simulation run in tabular format. The window is divided into two adjustable panes, which allow you to scroll horizontally through the listing on the right, while keeping time and delta visible on the left.




## HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

- *VHDL items*
  signals and process variables
- *Verilog items*
  nets and register variables

**Note:** Constants, generics, parameters, and memories are not viewable in the List or Wave windows.

## The List window menu bar

The following menu commands and button options are available from the List window menu bar.

### File menu

| | |
|---|---|
| Write List (format) | save the listing as a text file in one of three formats: tabular, events, or TSSI |
| Load Format | run a List window format DO file previously saved with Save Format |
| Save Format | saves the current List window display and signal preferences to a do (macro) file; running the DO file will reformat the List window to match the display as it appeared when the DO file was created |
| Close | close this copy of the List window |

### Edit menu

| | |
|---|---|
| Cut | cut the selected item field from the listing; see "Editing and formatting HDL items in the List window" (7-135) |
| Copy | copy the selected item field |
| Paste | paste the previously cut or copied item to the left of the currently selected item |
| Delete | delete the selected item field |
| Combine | combine the selected fields into a user-defined bus; keep copies of the original items rather than moving them; see "Combine signals into a user-defined bus" (7-110) |
| Select All | select all signals in the List window |
| Unselect All | deselect all signals in the List window |
| Find... | find specified item label within the List window |

### Markers menu

| | |
|---|---|
| Add Marker | add a time marker at the top of the listing page |
| Delete Marker | delete the selected marker from the listing |
| Goto | choose the time marker to go to from a list of current markers |

### Prop menu

| | |
|---|---|
| Display Props | set display properties for all items in the window: delta settings, trigger on selection, strobe period, and label size |
| Signal Props | set label, radix, trigger on/off, and field width for the selected item |

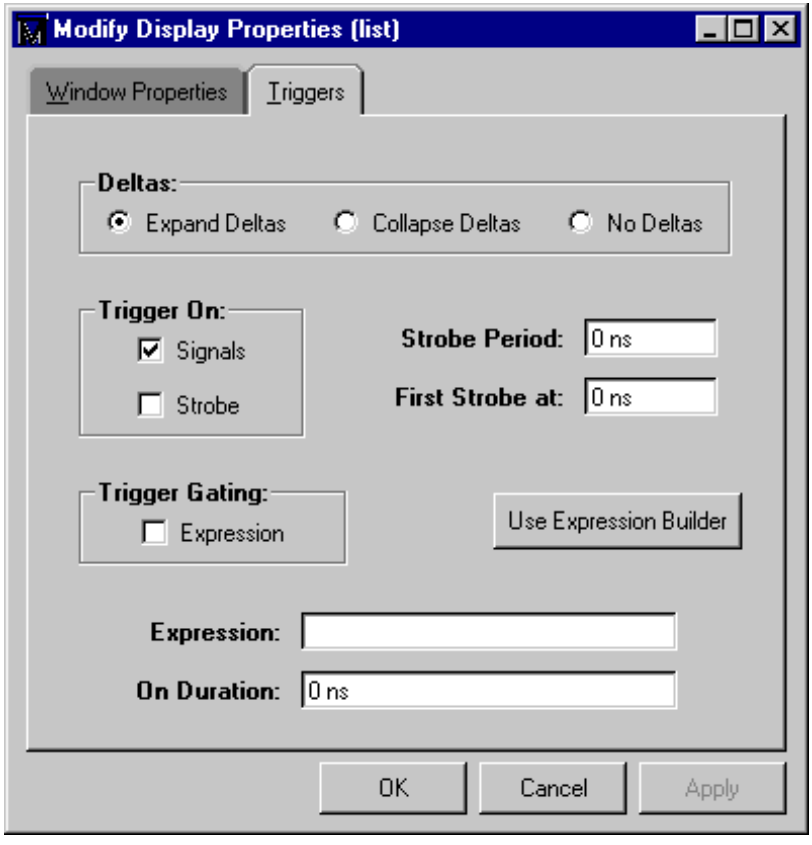
### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## Setting List window display properties

Before you add items to the List window you can set the window's display properties. To change when and how a signal is displayed in the List window, make this selection from the List window menu bar: **Prop > Display Props**. The resulting Modify Display Properties dialog box has the following options.

### Trigger settings page

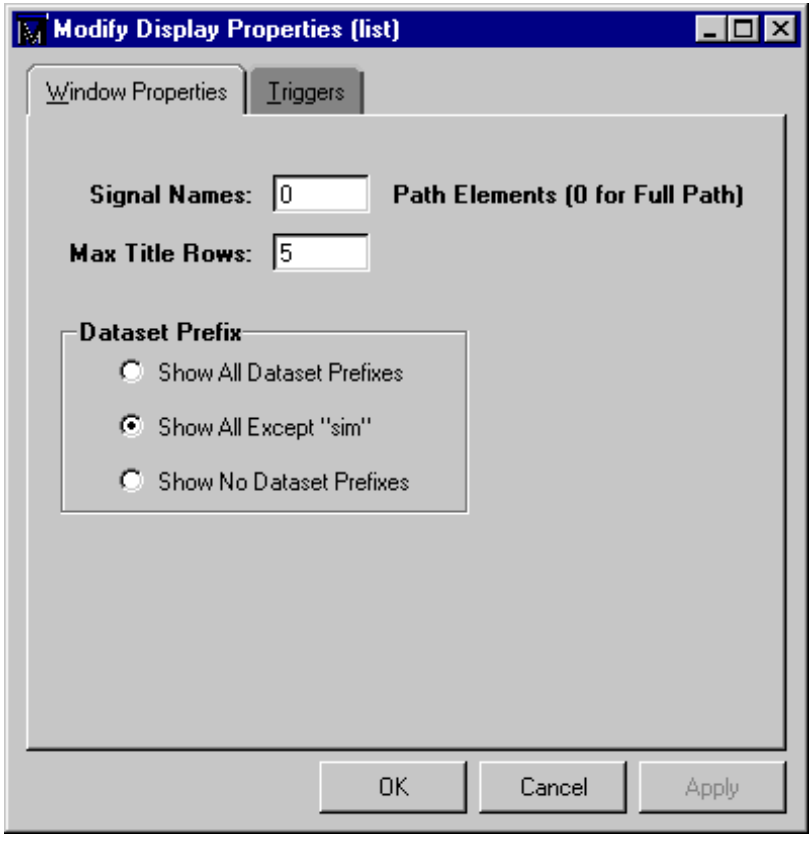



The Triggers page controls the triggering for the display of new lines in the List window. You can specify whether an HDL item trigger or a strobe trigger is used to determine when the List window displays a new line. If you choose **Trigger on: Signals**, then you can choose between collapsed or expanded delta displays. You can also choose a combination of signal or strobe triggers. To use gating, Signals or Strobe or both must be selected.

The Triggers page includes the following options:

- **Deltas:Expand Deltas**
  When selected with the **Trigger on: Signals** check box, displays a new line for each time step on which items change, including deltas within a single unit of time resolution.
- **Deltas:Collapse Deltas**
  Displays only the final value for each time unit in the List window.
- **Deltas:No Deltas**
  No simulation cycle (delta) column is displayed in the List window.
- **Trigger On: Signals**
  Triggers on signal changes. Defaults to all signals. Individual signals may be excluded from triggering by using the **Prop > Signals Props** dialog box or by originally adding them with the **-notrigger** option to the **add list** command (CR-19).
- **Trigger On: Strobe**
  Triggers on the **Strobe Period** you specify; specify the first strobe with **First Strobe at:**.
- **On Duration**
  The duration for gating to remain open after the last list row in which the expression evaluates to true; expressed in x number of default timescale units. Gating is level-sensitive rather than edge-triggered.

Window Properties page




The **Window Properties** page includes these options:

- **Signal Names**
Allows you to determine the number of path elements to be shown in the List window. For example, "0" shows the full path. "1" shows only the leaf element.
- **Max Title Rows**
The maximum number of rows in the name pane.
- **Dataset Prefix: Show All Dataset Prefixes**
Display the dataset prefix associated with each signal pathname. Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show All Except "sim"**
Display all dataset prefixes except the one associated with the current simulation – "sim." Useful for displaying signals from multiple datasets.
- **Dataset Prefix: Show No Dataset Prefixes**
Do not display any dataset prefixes.

## Adding HDL items to the List window

Before adding items to the List window you may want to set the window display properties (see "Setting List window display properties" (7-131)). You can add items to the List window in several ways.

### Adding items with drag and drop

You can drag and drop items into the List window from the Process, Signals, or Structure window. Select the items in the first window, then drop them into the List window. Depending on what you select, all items or any portion of the design may be added.

### Adding items from the Main window command line

Invoke the **add list** (CR-19) command to add one or more individual items; separate the names with a space:

```
add list <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list *
```

Or add all the items in the design with:

```
add list -r \*
```

### Adding items with a List window format file

To use a List window format file you must first save a format file for the design you are simulating. The saved format file can then be used as a DO file to recreate the List window formatting.

- add HDL items to your List window
- edit and format the items to create the view you want, see "Editing and formatting HDL items in the List window" (7-135)
- save the format to a file with the List window menu selection: **File > Save Format**

To use the format (do) file, start with a blank List window, and run the DO file in one of two ways:

- use the **do** (CR-43) command on the command line:
  `do <my_list_format>`
- select **File > Load Format** from the List window menu bar

Use **Edit > Select All** and **Edit > Delete** to remove the items from the current List window or create a new, blank List window with the **View > New > List** selection from the "Main window" (7-113). You may find it useful to have two differently formatted windows open at the same time, see "Examining simulation results with the List window" (7-138).

**Note:** List window format files are design-specific; use them only with the design you were simulating when they were created. If you try to the wrong format file, Model*Sim* will advise you of the HDL items it expects to find.

## Editing and formatting HDL items in the List window

Once you have the HDL items you want in the List window, you can edit and format the list to create the view you find most useful. (See also, "Adding HDL items to the List window" (7-134))

### To edit an item:

Select the item's label at the top of the List window or one of its values from the listing. Move, copy or remove the item by selecting commands from the List window Edit menu (7-129) menu.

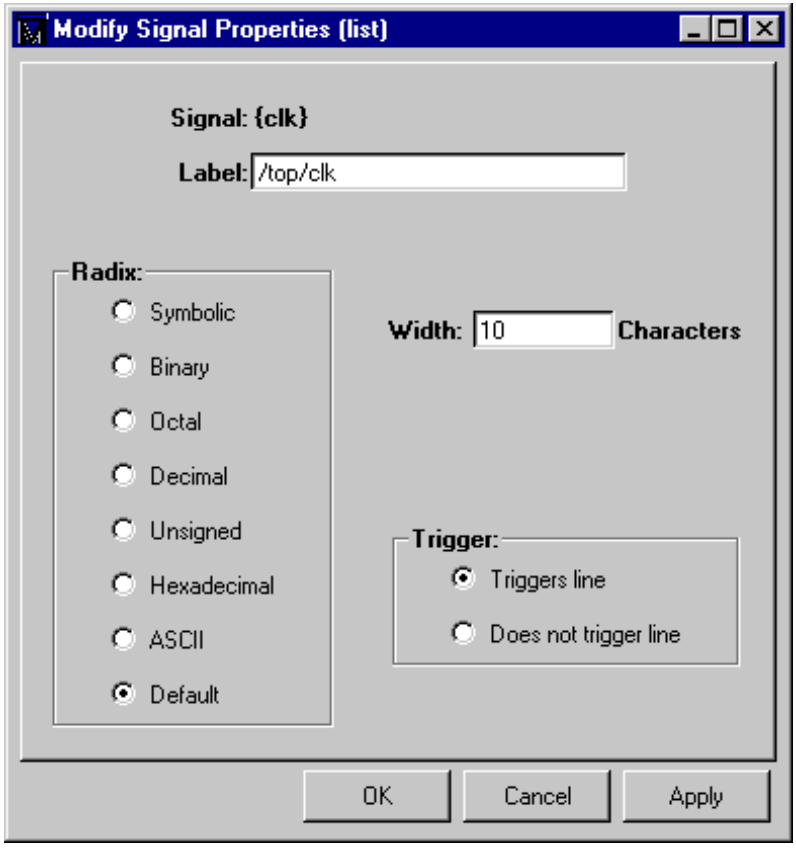
You can also click+drag to move items within the window:

- to select several contiguous items:
  click+drag to select additional items to the right or the left of the original selection
- to select several items randomly:
  Control+click to add or subtract from the selected group
- to move the selected items:
  re-click on one of the selected items, hold and drag it to the new location

### To format an item:

Select the item's label at the top of the List window or one of its values from the listing, then use the **Prop > Signal Props** menu selection. The resulting Modify

Signal Properties dialog box allows you to set the item's label, label width, triggering, and radix.




The **Modify Signal Properties** dialog box includes these options:

- **Signal**
Shows the signal you selected with the mouse with its dataset prefix.
- **Label**
Allows you to specify the label that is to appear at the top of the List window column for the specified item.
- **Radix**
Allows you to specify the radix (base) in which the item value is expressed. The default radix is symbolic, which means that for an enumerated type, the List window lists the actual values of the enumerated type of that item.

For the other radixes - binary, octal, decimal, unsigned, hexadecimal, or ASCII - the item value is converted to an appropriate representation in that radix. In the system initialization file, *modelsim.tcl*, you can specify the list translation rules for arrays of enumerated types for binary, octal, decimal, unsigned decimal, or hexadecimal item values in the design unit.

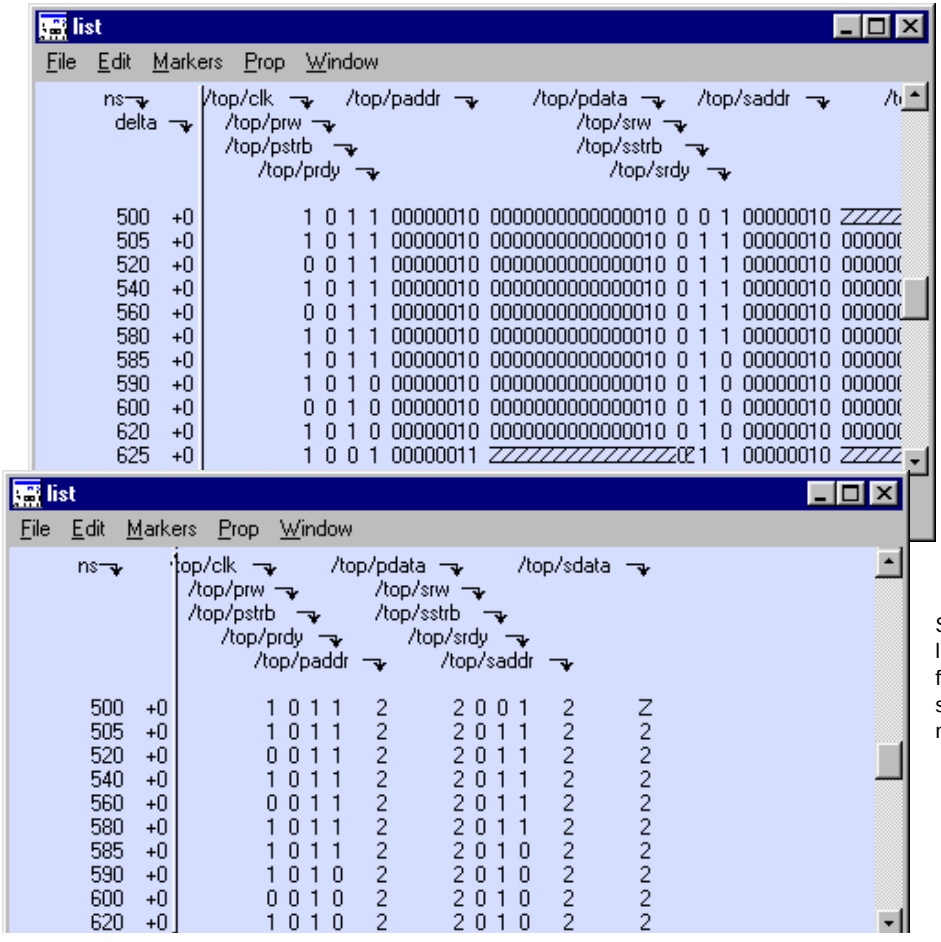
- **Width**
Allows you to specify the desired width of the column used to list the item value. The default is an approximation of the width of the current value.

- **Trigger: Triggers line**
  Specifies that a change in the value of the selected item causes a new line to be displayed in the List window.
- **Trigger: Does not trigger line**
  Selecting this option in the List Signals window specifies that a change in the value of the selected item does not affect the List window.

The trigger specification affects the trigger property of the selected item. See also, "Setting List window display properties" (7-131).

## Examining simulation results with the List window

Because you can use the Main window View menu (7-116) to create a second List window, you can reformat another List window after the simulation run if you decide a different format would reveal the information you're after. Compare the two illustrations.



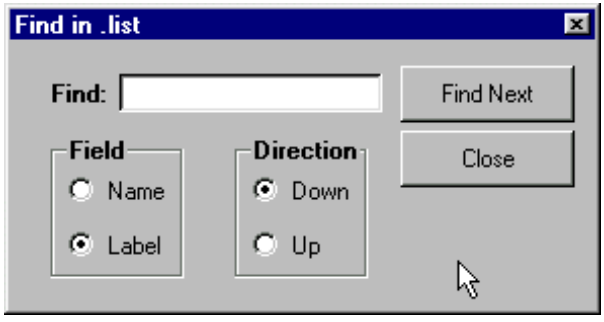

The divider bar separates resolution and delta from values; signal values are listed in symbolic format; and an item change triggers a new line.

Signal values are listed in decimal format; a 20ns strobe triggers a new line

In the first List window, the HDL items are formatted as symbolic and use an item change to trigger a line; the field width was changed to accommodate the default label width. The window divider maintains the time and delta in the left pane; signals in the right pane may be viewed by scrolling. For the second listing, the specification for triggering was changed to a 100-ns strobe, and the item radix for **a**, **b**, **cin**, and **sum** is now decimal.

## Finding items by name in the List window

The Find dialog box allows you to search for text strings in the List window. From the List window select **Edit > Find** to bring up the Find dialog box.

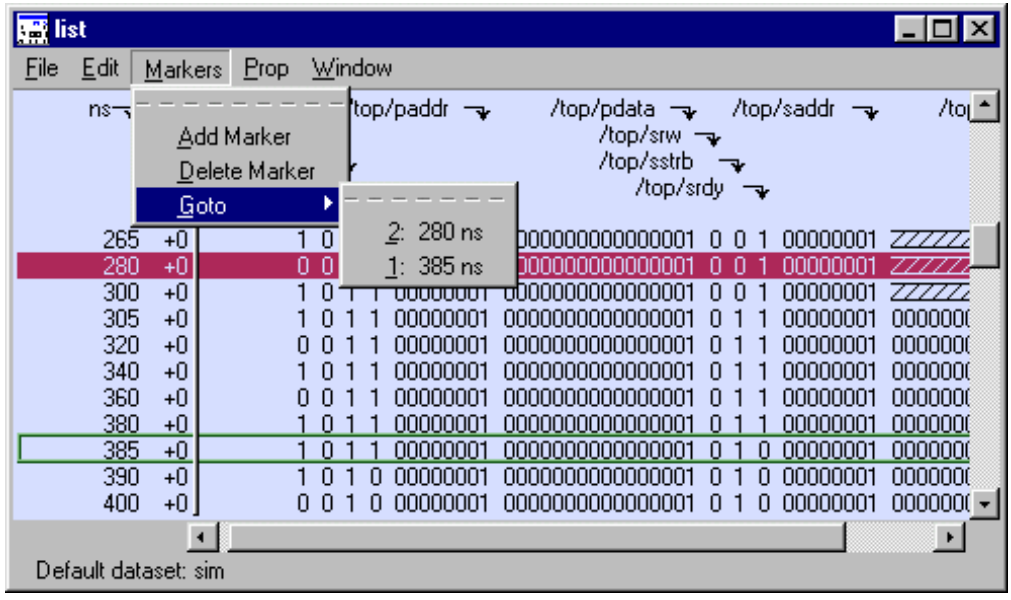



Enter an item label and **Find** it by searching **Forward** (right) or **Reverse** (left) through the List window display. The column number of the item found displays at the bottom of the dialog box. Note that you can change an item's label, see "Setting List window display properties" (7-131).

## Setting time markers in the List window

From the List window select **Markers > Add Marker** to tag the selected list line with a marker. The marker is indicated by a thin box surrounding the marked line. The selected line uses the same indicator, but its values are highlighted. Delete markers by first selecting the marked line, then making the **Markers > Delete Marker** menu selection.

Finding a marker




Choose a specific marked line to view with **Markers > Goto** menu selection.The marker name (on the **Goto** list) corresponds to the simulation time of the selected line.

## List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

| Key | Action |
|---|---|
| <arrow up> | scroll listing up |
| <arrow down> | scroll listing down |
| <arrow left> | scroll listing left |
| <arrow right> | scroll listing right |
| <page up> | scroll listing up by page |
| <page down> | scroll listing down by page |

| Key | Action |
|---|---|
| <tab> | searches forward (down) to the next transition on the selected signal |
| <shift-tab> | searches backward (up) to the previous transition on the selected signal (does not function on HP workstations) |
| <control-f> | opens the find dialog box; find the specified item label within the list display |

## Saving List window data to a file

From the List window select **Edit > Write List (format)** to save the List window data in one of these formats:

- **tabular**
  writes a text file that looks like the window listing

```
ns   delta      /a        /b      /cin     /sum     /cout
0      +0        X         X        U        X        U
0      +1        0         1        0        X        U
2      +0        0         1        0        X        U
```

- **event**
  writes a text file containing transitions during simulation

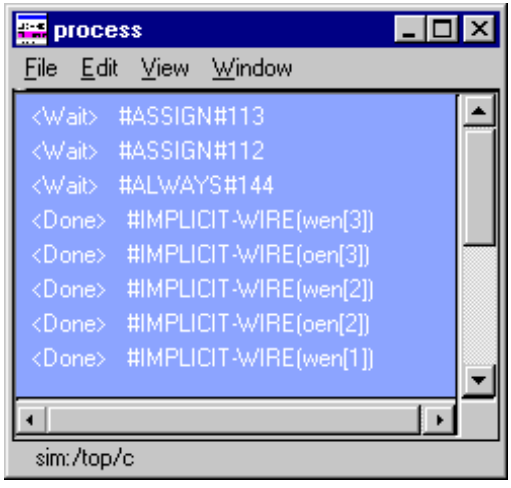
```
@0 +0
/a X
/b X
/cin U
/sum X
/cout U
@0 +1
/a 0
/b 1
/cin 0
```

- **TSSI**
  writes a file in standard TSSI format; see also, the **write tssi** command (CR-172)

```
0 0000000000000010????????
2 0000000000000010???????1?
3 0000000000000010??????010
4 0000000000000010000000010
100 0000001000000010000000010
```

## Process window

The Process window displays a list of processes (either active or in region) and indicates the pathname of the instance in which the process is located.




Each HDL item in the scrollbox is preceded by one of the following indicators:

- **<Ready>**
Indicates that the process is scheduled to be executed within the current delta time.
- **<Wait>**
Indicates that the process is waiting for a VHDL signal or Verilog net or variable to change or for a specified time-out period.
- **<Done>**
Indicates that the process has executed a VHDL wait statement without a time-out or a sensitivity list. The process will not restart during the current simulation run.

If you select a "Ready" process, it will be executed next by the simulator.

When you click on a process in the Process window the following windows are updated:

| Window updated | Result |
|---|---|
| Structure window (7-158) | shows the region in which the process is located |
| Variables window (7-161) | shows the VHDL variables and Verilog register variables in the process |
| Source window (7-152) | shows the associated source code |
| Dataflow window (7-124) | shows the process, the signals and nets the process reads, and the signals and nets driven by the process. |

## The Process window menu bar

The following menu commands and button options are available from the Process window menu bar.

### File menu

| | |
|---|---|
| Save As | save the process tree to a text file viewable with the Model*Sim* **notepad** (CR-70) |
| Environment | Follow Process Selection: update the window based on the selection in the Structure window (7-158); Fix to Process: maintain the current view, do not update |
| Close | close this copy of the Process window |

### Edit menu

| | |
|---|---|
| Copy | copy the selected process |
| Sort | sort the process list in either ascending, descending, or declaration order |
| Select All | select all signals in the Process window |
| Unselect All | deselect all signals in the Process window |
| Find... | find specified text string within the structure tree; choose the Status (ready, wait or done) or Process label to search and the search direction: forward or reverse |

### View menu

| | |
|---|---|
| Active | Displays all the processes that are scheduled to run during the current simulation cycle. |
| In Region | Displays any processes that exist in the region that is selected in the Structure window. |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |

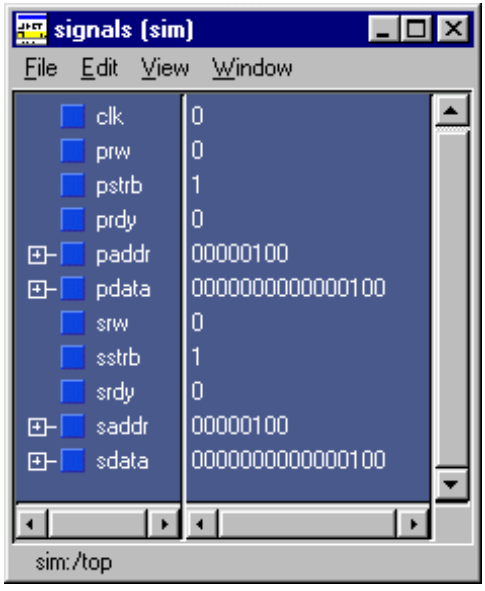
| | |
|---|---|
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## Signals window

The Signals window is divided into two window panes. The left pane shows the names of HDL items in the current region (which is selected in the Structure window). The right pane shows the values of the associated HDL item pathname at the end of the current run. The data in this pane is similar to that shown in the Wave window (7-164), except that the values do not change dynamically with movement of the select Wave window cursor.

Horizontal scroll bars for each window pane allow scrolling to the right or left in each pane individually. The vertical scroll bar will scroll both panes together.

The HDL items may be sorted in ascending, descending, or declaration order.




### HDL items you can view

One entry is created for each of the following VHDL and Verilog HDL items within the design:

*VHDL items*

signals

*Verilog items*

nets, register variables, and named events

*Virtual items*

(indicated by an orange diamond icon) virtual signals, see "Virtual signals" (6-101) for more information

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion.

Hierarchy also applies to Verilog nets and vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.)

Hierarchy is indicated in typical Model*Sim* fashion with plus (expandable), minus (expanded), and blank (single level) boxes.

See "Tree window hierarchical view" (7-111) for more information.

## The Signals window menu bar

The following menu commands are available from the Signals window menu bar.

### File menu

| | |
|---|---|
| Save As | save the signals tree to a text file viewable with the Model*Sim* **notepad** (CR-70) |
| Environment | Follow Environment: update the window based on the selection in the Structure window (7-158); Fix to Context: maintain the current view, do not update |
| Close | close this copy of the Signals window |

### Edit menu

| | |
|---|---|
| Copy | copy the current selection in the Signals window |
| Sort | sort the signals tree in either ascending, descending, or declaration order |
| Select All | select all items in the Signals window |
| Unselect All | unselect all items in the Signals window |
| Expand Selected | expands the hierarchy of the selected item |
| Collapse Selected | collapses the hierarchy of the selected item |
| Expand All | expands the hierarchy of all items that can be expanded |
| Collapse All | collapses the hierarchy of all expanded items |
| Force... | apply stimulus to the specified Signal Name; specify Value, Kind (Freeze/Drive/Deposit), Delay, and Repeat; see also the **force** command (CR-56) |
| Noforce | removes the effect of any active **force** command (CR-56) on the selected HDL item; see also the **noforce** command (CR-67) |
| Clock... | defines clock signals by Signal Name, Period, Duty Cycle, Offset, and whether the first edge is rising or falling, see"Defining clock signals" (7-151) |
| Justify Values | justify values to the left or right margins of the window pane |
| Find... | find specified text string within the Signals window; choose the Name or Value field to search and the search direction: forward or reverse**searchlog** command (CR-88) |

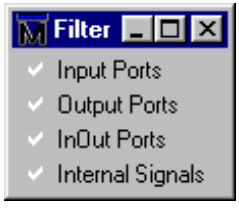
### View menu

| | |
|---|---|
| Wave/List/Log | place the Selected Signals, Signals in Region, or Signals in Design in the Wave window (7-164), List window (7-128), or logfile |
| Filter | choose the port and signal types to view (Input Ports, Output Ports, InOut Ports and Internal Signals) in the Signals window |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## Selecting HDL item types to view

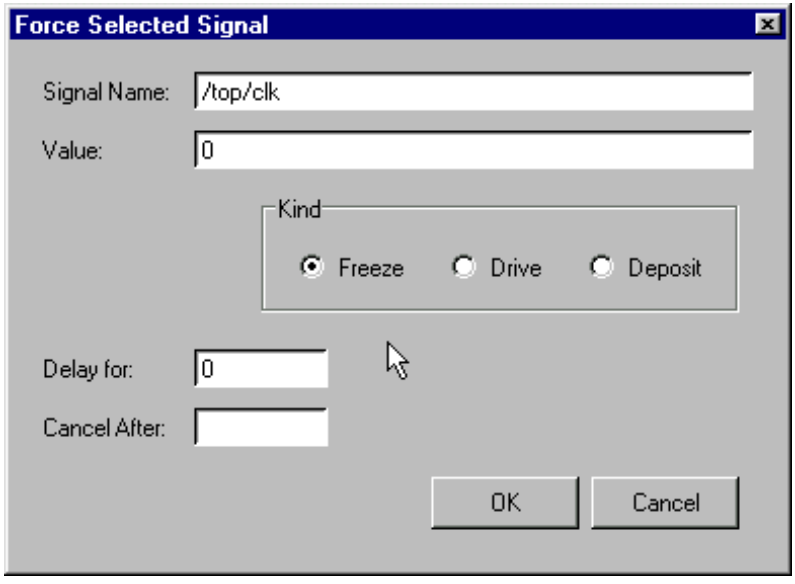



The **View > Filter...** menu selection allows you to specify which HDL items are shown in the Signals window. Multiple options may be selected.

## Forcing signal and net values

The **Edit > Force** menu selection displays a dialog box that allows you to apply stimulus to the selected signal or net. You can specify that the stimulus is to repeat at a regular time interval, expressed in the time units set in the Startup window

when you invoked the simulator. Multiple signals may be selected and forced; the force dialog box remains open until all of the signals are either forced, skipped, or you close the dialog box. See also the **force** command (CR-56).




The **Force** dialog box includes these options:

- **Signal Name**
Specify the signal or net for the applied stimulus.

- **Value**
Initially displays the current value, which can be changed by entering a new value into the field. A value can be specified in radixes other than decimal by using the form (for VHDL and Verilog, respectively):

```
base#value -or-
b|o|d|h'value
```

16#EE or h'EE, for example, specifies the hexadecimal value EE.

- **Kind: Freeze**
Freezes the signal or net at the specified value until it is forced again or until it is unforced with a **noforce** command (CR-67).

  **Freeze** is the default for Verilog nets and unresolved VHDL signals and **Drive** is the default for resolved signals.

  If you prefer **Freeze** as the default for resolved and unresolved signals, you can change the default force kind in the *modelsim.ini* file; see "Projects and system initialization" (3-35).

- **Kind: Drive**
  Attaches a driver to the signal and drives the specified value until the signal or net is forced again or until it is unforced with a **noforce** command (CR-67). This value is illegal for unresolved VHDL signals.
- **Kind: Deposit**
  Sets the signal or net to the specified value. The value remains until there is a subsequent driver transaction, or until the signal or net is forced again, or until it is unforced with a **noforce** command (CR-67).
- **Delay for**
  Allows you to specify how many time units from the current time the stimulus is to be applied.
- **Cancel After**
  Cancels the **force** command (CR-56) after the specified period of simulation time.
- **OK**
  When you click the OK button, a **force** command (CR-56) is issued with the parameters you have set, and is echoed in the Main window. If more than one signal is selected to force, the next signal down appears in the dialog box each time the OK button is selected. Unique force parameters may be set for each signal.

## Adding HDL items to the Wave and List windows or a logfile

Before adding items to the List or Wave window you may want to set the window display properties (see "Setting List window display properties" (7-131)). Once display properties have been set, you can add items to the windows or logfile in several ways.

### Adding items from the Main window command line

Use the **View** menu with either the **Wave**, **List**, or **Log** selection to add HDL items to the Wave window (7-164), Saving the Dataflow window as a Postscript file (7-127)or a logfile, respectively.

The logfile is written as an archive file in binary format and is used to drive the List and Wave window at a later time. Once signals are added to the logfile they cannot be removed. If you begin a simulation by invoking **vsim** (CR-148) with the -view <logfile_name> option, VSIM reads the logfile to drive the Wave and List windows.

Choose one of the following options (Model*Sim* opens the target window for you):

- **Selected signal**

Lists only the item(s) selected in the Signals window.

- **Signals in region**

Lists all items in the region that is selected in the Structure window.

- **Signals in design**

Lists all items in the design.

### Adding items from the Main window command line

Another way to add items to the Wave or List window or the logfile is to enter the one of the following commands at the VSIM prompt (choose either the **add list** (CR-19), or **log** (CR-62) command):

```
add list | add wave | log <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add list | add wave | log *
```

Or add all the items in the design with:

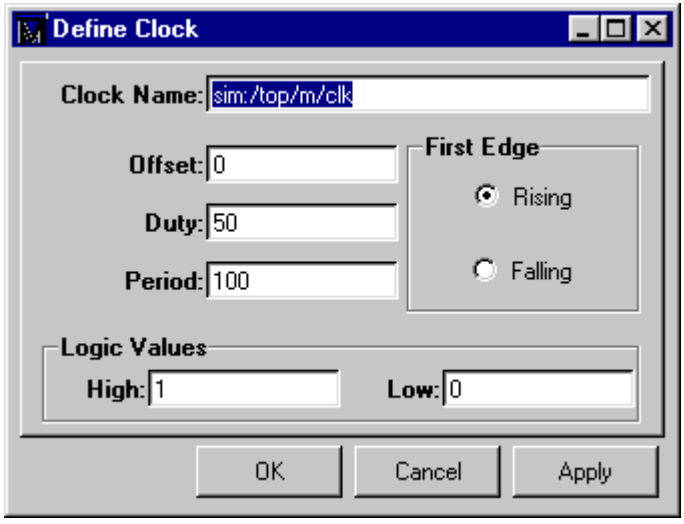
```
add list | add wave | log -r /*
```

If the target window (Wave or List) is closed, Model*Sim* opens it when you when you invoke the command.

## Finding HDL items in the Signals window

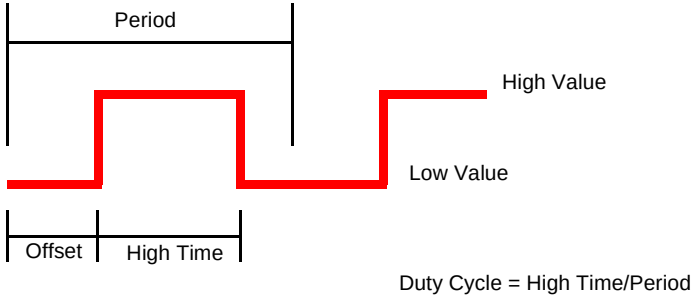
Find the specified text string within the Signals window; choose the **Name** or **Value** field to search and the search direction: **Forward** or **Reverse**.

## Defining clock signals




Selecting Clock from the Edit menu allows you to define clock signals by Name, Period, Duty Cycle, Offset, and whether the first rising edge is rising or falling.

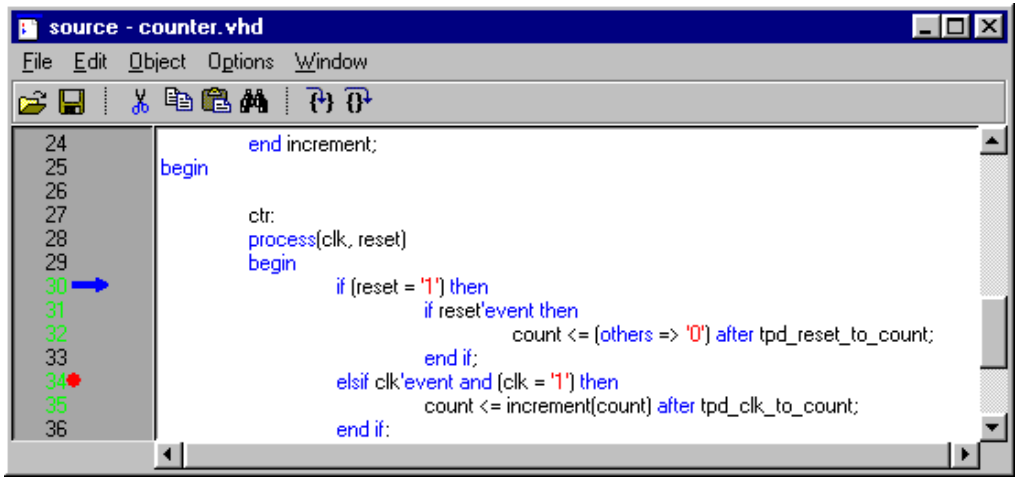
For clock signals starting on the rising edge, the definition for Period, Offset, and Duty Cycle is as follows:




If the signal type is std_logic, std_ulogic, bit, verilog wire, verilog net, or any other logic type where 1 and 0 are valid, then 1 is the default High Value and 0 is the default Low Value. For other signal types, you will need to specify a High Value and a Low Value for the clock.

## Source window

The Source window allows you to view and edit your HDL source code. Select an item in the Structure window (7-158) or use the **File** menu to add a source file to the window, then select a process in the Process window (7-142) to view that process; an arrow next to the line numbers indicates the selected process. (Your source code can remain hidden if you wish, see "Source code security and -nodebug" (C-292). A dot next to a line number indicates a breakpoint. Breakpoints can only occur at executable lines, which are indicated by green line numbers.




If any breakpoints have been set, each is signified by a colored dot next to a line number at the left side of the window pane. To set a breakpoint, click at or near the line number in the numbered area at the left side of the window. The breakpoints are toggles, so you can click again to delete an existing breakpoint. There is no limit to the number of breakpoints you can set. See also the **bp** command (CR-31) (breakpoint) command.

To look at a file that is not currently being displayed, use the Structure window (7-158) to select a different design unit or use the Source menu selection: **File** > **Open**. The pathname of the source file is indicated in the header of the Source window.

You can copy and paste text between the Source window and the Main window (7-113); select the text you want to copy, then paste it into the Main window with the middle button (3-button mouse), right button (2-button mouse).

## The Source window menu bar

The following menu commands are available from the Source window menu bar.

### File menu

| | |
|---|---|
| New | edit a new (VHDL, Verilog or Other) source file |
| Open | select a source file to open |
| Use Source | specifies an alternative file to use for the current source file; this alternative source mapping exists for the current simulation only |
| Source Directory | add to a list of directories (the SourceDir variable in modelsim.tcl) to search for source files |
| Properties | lists a variety of information about the source file; for example, file type, file size, file modification date |
| Save | save the current source file |
| Save_As | save the current source file with a different name |
| Compile | complies HDL source files |
| Close | close this copy of the Source window |

### Edit menu

To edit a source file, make sure the **Read Only** option in the Source Options dialog box is *not* selected (use the Source menu **Edit > read only** selection).

| | |
|---|---|
| <editing option> | basic editing options include: Cut, Copy, Paste, Select All, and Unselect All; see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |
| Find... | find the specified text string within the source file; match case option |
| read only | toggles the read-only status of the current source file |

### Object menu

| | |
|---|---|
| Describe | displays information about the selected HDL item; same as the **describe** command (CR-41); the item name is shown in the title bar |
| Examine | displays the current value of the selected HDL item; same as the **examine** (CR-51) command; the item name is shown in the title bar |

### Options menu

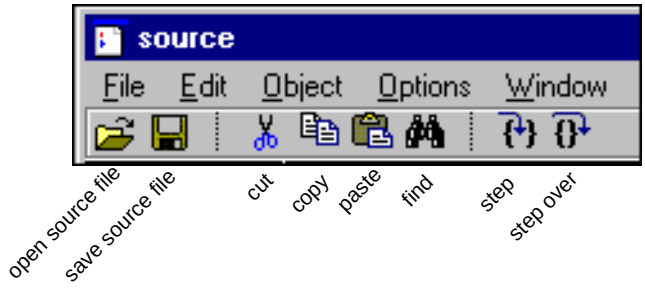
| | |
|---|---|
| Options | open the Source Options dialog box, see "Setting Source window options" (7-157) |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## The Source window tool bar

Buttons on the Source window tool bar gives you quick access to these Model*Sim* commands and functions.




| Source window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Other equivalents** |
| | **Open Source File**<br>open the Open dialog box (you can open any text file for editing in the Source window) | File > Open | select an HDL item in the Structure window, the associated source file is loaded into the Source window |
| | **Save Source File**<br>save the file in the Source window | File > Save | none |
| | **Cut**<br>cut the selected text within the Source window | Edit > Cut | see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |
| | **Copy**<br>copy the selected text within the Source window | Edit > Copy | see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |

| Source window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Other equivalents** |
| | **Paste**<br>paste the copied text to the cursor location | Edit > Paste | see: "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) |
| | **Find**<br>find the specified text string within the source file; match case option | Edit > Find | none |
| | **Step**<br>steps the current simulation to the next HDL statement | none | use **step** command at the VSIM prompt<br><br>see: **step** (CR-93) command |
| | **Step Over** | none | use the **step -over** command at the VSIM prompt<br><br>see: **step** (CR-93) command |

## Editing the source file in the Source window

Several tool bar buttons (shown above), mouse actions, and special keystrokes can be used to edit the source file in the Source window. See "Mouse and keyboard shortcuts in the Transcript and Source windows" (7-121) for a list of mouse and keyboard editing options.

## Checking HDL item values and descriptions

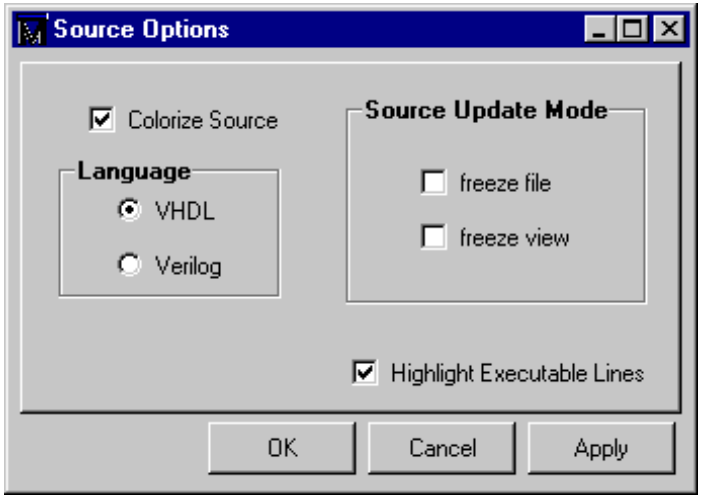
There are two quick methods to determine the value and description of an HDL item displayed in the Source window:

- select an item, then chose **Object > Examine** or **Object > Description** from the Source window menu
- select an item with the right mouse button to view examine pop-up (select "now" to examine the current simulation time in VHDL code)

You can also invoke the **examine** (CR-51) and/or **describe** (CR-41) command on the command line or in a macro.

## Setting Source window options

Access the Source window options with this Source menu selection: **Options** > **Options**.



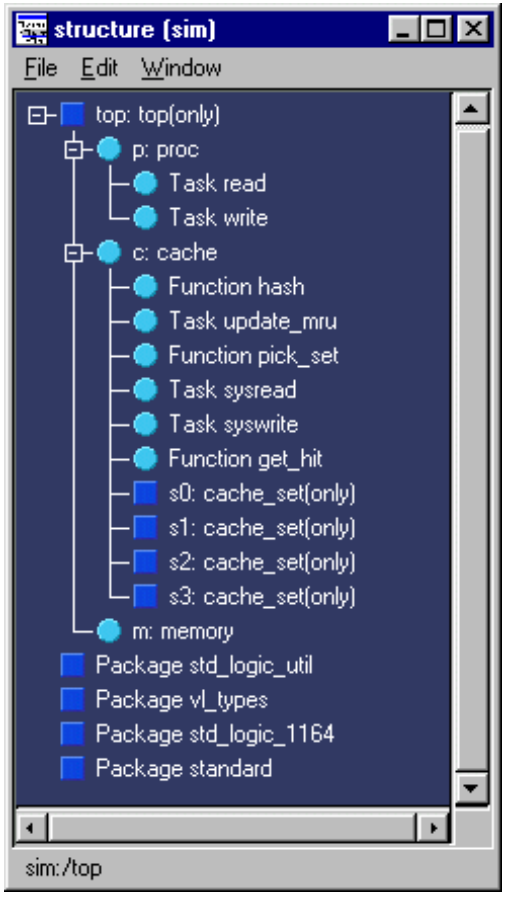

The **Source Options** dialog box includes these options:

- **Language**
select either **VHDL** or **Verilog**; sets language for key word colorizing
- **Source Update Mode**
select **freeze file** to maintain the same source file in the Source window (useful when you have two Source windows open; one can be updated from the Structure window (7-158), the other frozen) or **freeze view** to disable updating the source view from the Process window (7-142)

- **Colorize Source**
colorize key words, variables and comments
- **Highlight Executable Lines**
highlights the line number of executable lines

# Structure window

The Structure window provides a hierarchical view of the structure of your design. An entry is created by each HDL item within the design. (Your design structure can remain hidden if you wish, see "Source code security and -nodebug" (C-292).)




### HDL items you can view

The following HDL items for VHDL and Verilog are represented by hierarchy within Structure window.

*VHDL items*

(indicated by a dark blue square icon) signals, variables, component instantiation, generate statement, block statement, and package

*Verilog items*

(indicated by a lighter blue circle icon) parameters, registers, nets, module instantiation, named fork, named begin, task, and function

*Virtual items*

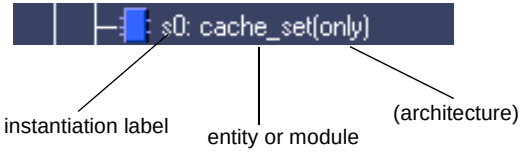
(indicated by an orange diamond icon) virtual signals, buses, and functions, see "Virtual Objects (User-defined buses, and more)" (6-100) for more information.

You can expand and contract the display to view the hierarchical structure by clicking on the boxes that contain "+" or "-". Clicking "+" expands the hierarchy so the sub-elements of that item can be seen. Clicking "-" contracts the hierarchy.

The first line of the Structure window indicates the top-level design unit being simulated. By default, this is the only level of the hierarchy that is expanded upon opening the Structure window.

### Instance name components in the Structure window

An instance name displayed in the Structure window consists of the following parts:




where:

- **instantiation label**
  Indicates the label assigned to the component or module instance in the instantiation statement.
- **entity or module**
  Indicates the name of the entity or module that has been instantiated.
- **architecture**
  Indicates the name of the architecture associated with the entity (not present for Verilog).

When you select a region in the Structure window, it becomes the *current region* and is highlighted; the Source window (7-152) and Signals window (7-144) change dynamically to reflect the information for that region. This feature provides a useful method for finding the source code for a selected region because the system keeps track of the pathname where the source is located and displays it automatically, without the need for you to provide the pathname.

Also, when you select a region in the Structure window, the Process window (7-142) is updated if **In Region** is selected in that window; the Process window will in turn update the Variables window (7-161).

## The Structure window menu bar

The following menu commands are available from the Structure window menu bar.

### File menu

| | |
|---|---|
| Save_As | save the structure tree to a text file viewable with the Model*Sim* **notepad** (CR-70) |
| Environment | changing of current environment between open datasets; or, establish a New Context by opening a new dataset |
| Close | close this copy of the Structure window |

### Edit menu

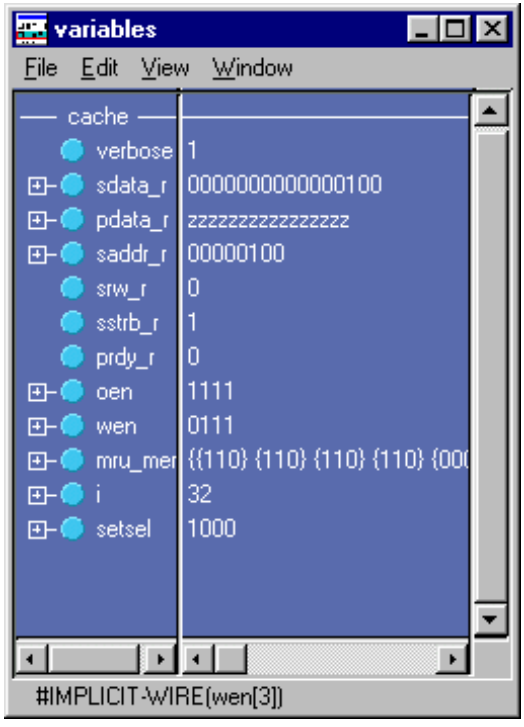
| | |
|---|---|
| Copy | copy the current selection in the Structure window |
| Sort | sort the structure tree in either ascending, descending, or declaration order |
| Expand Selected | expands the hierarchy of the selected item |
| Collapse Selected | collapses the hierarchy of the selected item |
| Expand All | expands the hierarchy of all items that can be expanded |
| Collapse All | collapses the hierarchy of all expanded items |
| Find... | find specified text string within the structure tree; choose the label for instance, entity/module or architecture to search for and the search direction: forward or reverse |

### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

# Variables window

The Variables window is divided into two window panes. The left pane lists the names of HDL items within the current process. The right pane lists the current value(s) associated with each name. The pathname of the current process is displayed at the bottom of the window. (The internal variables of your design can remain hidden if you wish, see "Source code security and -nodebug" (C-292).)




### HDL items you can view

The following HDL items for VHDL and Verilog are viewable within the Variables window.

*VHDL items*

constants, generics, and variables

*Verilog items*

register variables

The names of any VHDL composite types (arrays and record types) are shown in a hierarchical fashion. Hierarchy also applies to Verilog vector memories. (Verilog vector registers do not have hierarchy because they are not internally represented as arrays.) Hierarchy is indicated in typical Model*Sim* fashion with plus (expandable) and minus (expanded). See "Tree window hierarchical view" (7-111) for more information.

To change the value of a VHDL variable, constant, generic or Verilog register variable, move the pointer to the desired name and click to highlight the selection. Then select **Edit > Change** from the Variables window menu. This brings up a dialog box that lets you specify a new value. Note that "Variable Name" is a term that is used loosely in this case to signify VHDL constants and generics as well as VHDL and Verilog register variables. You can enter any value that is valid for the variable. An array value must be specified as a string (without surrounding quotation marks). To modify the values in a record, you need to change each field separately.

Click on a process in the Process window to change the Variables window.

## The Variables window menu bar

The following menu commands are available from the Variables window menu bar.

### File menu

| | |
|---|---|
| Save As | save the variables tree to a text file viewable with the Model*Sim* **notepad** (CR-70) |
| Environment | Follow Process Selection: update the window based on the selection in the Structure window (7-158); Fix to Process: maintain the current view, do not update |
| Close | close this copy of the Variables window |

### Edit menu

| | |
|---|---|
| Copy | copy the selected items in the Variables window |
| Sort | sort the variables tree in either ascending, descending, or declaration order |
| Select All | select all items in the Variables window |
| Unselect All | deselect all items in the Variables window |
| Expand Selected | expands the hierarchy of the selected item |
| Collapse Selected | collapses the hierarchy of the selected item |
| Expand All | expands the hierarchy of all items that can be expanded |
| Collapse All | collapses the hierarchy of all expanded items |
| Change | change the value of the selected HDL item |
| Justify Values | justify values to the left or right margins of the window pane |
| Find... | find specified text string within the variables tree; choose the Name or Value field to search and the search direction: forward or reverse |

### View menu

| | |
|---|---|
| Wave/List/Log | place the Selected Variables or Variables in Region in the Wave window (7-164), List window (7-128), or logfile |

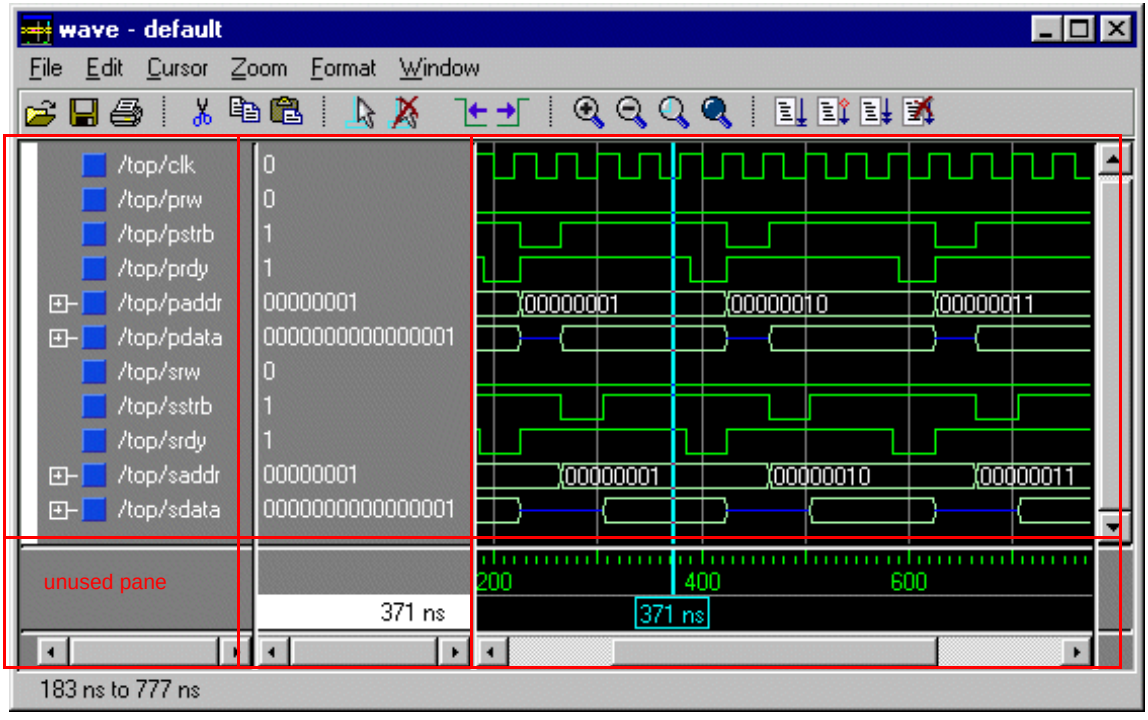
### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

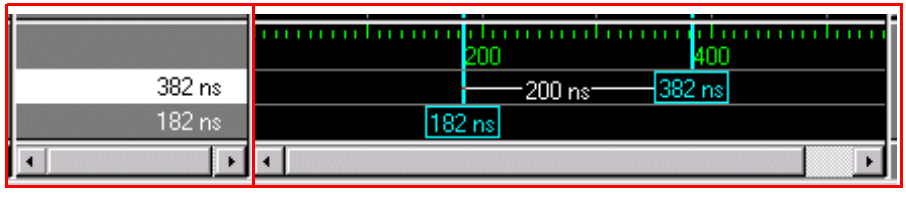
## Wave window

The Wave window, like the List window, allows you to view the results of your simulation. In the Wave window, however, you can see the results as HDL waveforms and their values.

The Wave window is divided into a number of window panes.



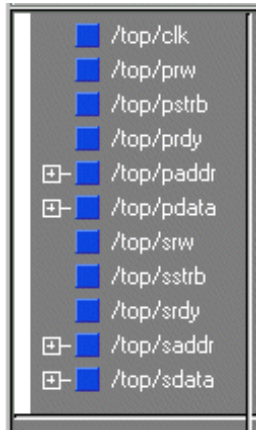

There are two cursor panes, as shown below. The left pane shows the time value for each cursor. The selected cursor's value is highlighted. The right pane shows the absolute time value for each cursor and relative time between cursors. Up to 20 cursors may be displayed.



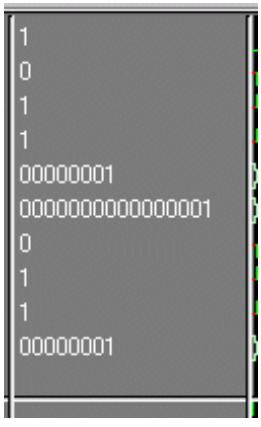

two cursor panes

The pathname pane displays signal pathnames. Signals may be displayed with full pathnames, as shown here, or with only the leaf element displayed. The selected signal is highlighted.

The white bar along the left margin indicates the selected Waveset (see Wave window panes (7-167)).




pathnames pane

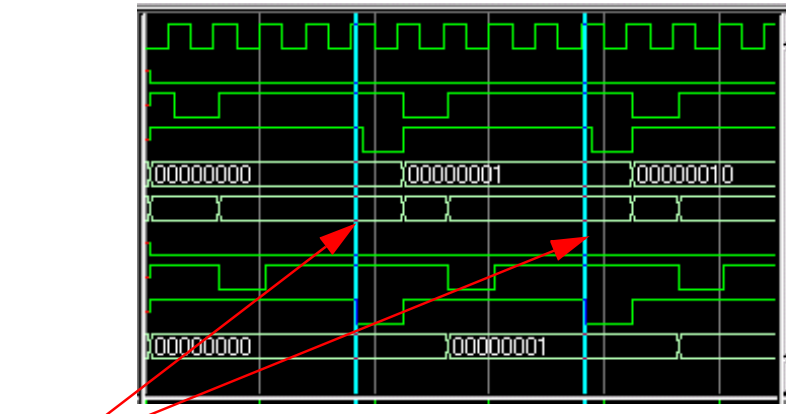



values pane

A values pane displays the values of the displayed signals. Signal values may be displayed in analog step, analog interpolated, analog backstep, literal, logic, and event formats. Each signal may be formatted individually. The default format is logic.

The radix for each signal may be symbolic, binary, octal, decimal, unsigned, hexadecimal, ASCII or default. The default radix may be set by selecting **File > Options > Simulation** in the Main window (see "Setting default simulation options" (7-211)).

The data in this pane is similar to that shown in the Signals window (7-144), except that the values change dynamically whenever a cursor in the waveform pane (below) is moved.




The waveform pane displays the waveforms that correspond to the displayed signal pathnames. It also displays up to 20 cursors.

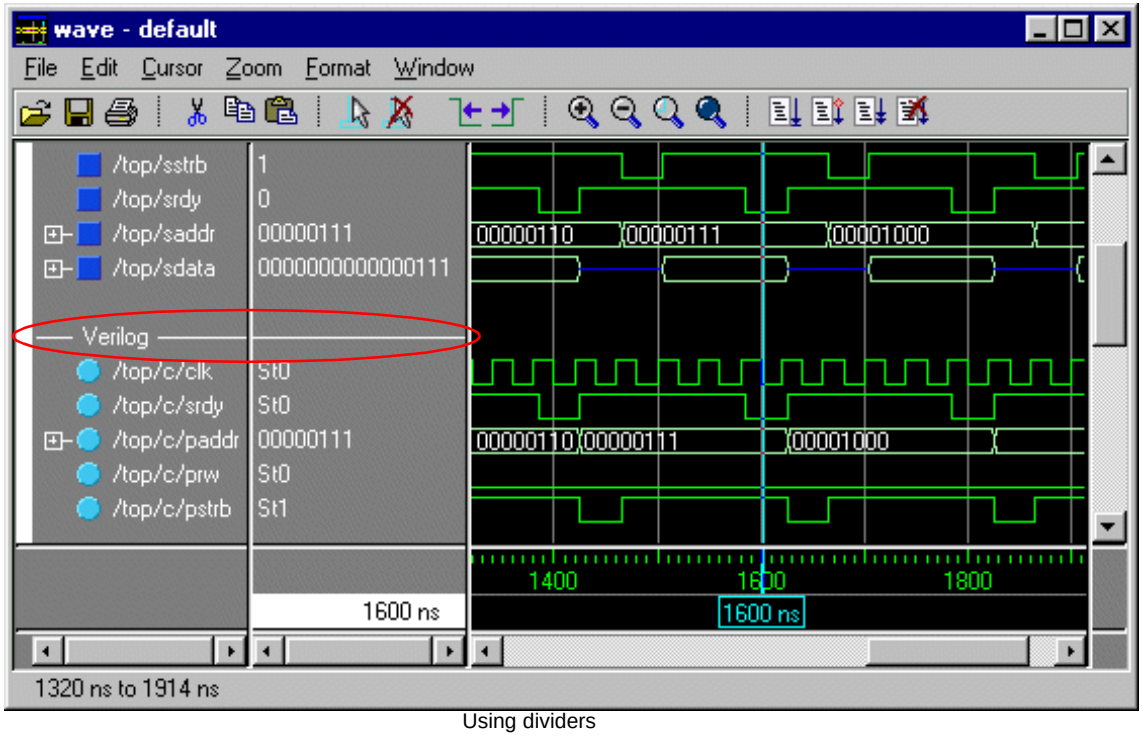
The window pane below the pathnames window pane and to the left of the cursor panes is unused at this time.

All window panes in the Wave window may be resized by clicking and dragging the bar between any two panes.

### Using Dividers

Dividing lines may be placed in the pathname and values window panes by selecting **File > New Divider**. Dividers serve as a visual aid to signal debugging, allowing you to separate signals and waveforms for easier viewing.

Dividing lines can be assigned any name, or no name at all. The default name is "New Divider." In the illustration below, VHDL signals have been separated from Verilog signals with at Divider called "Verilog." Notice that the waveforms in the waveform window pane have been separated by the divider as well.



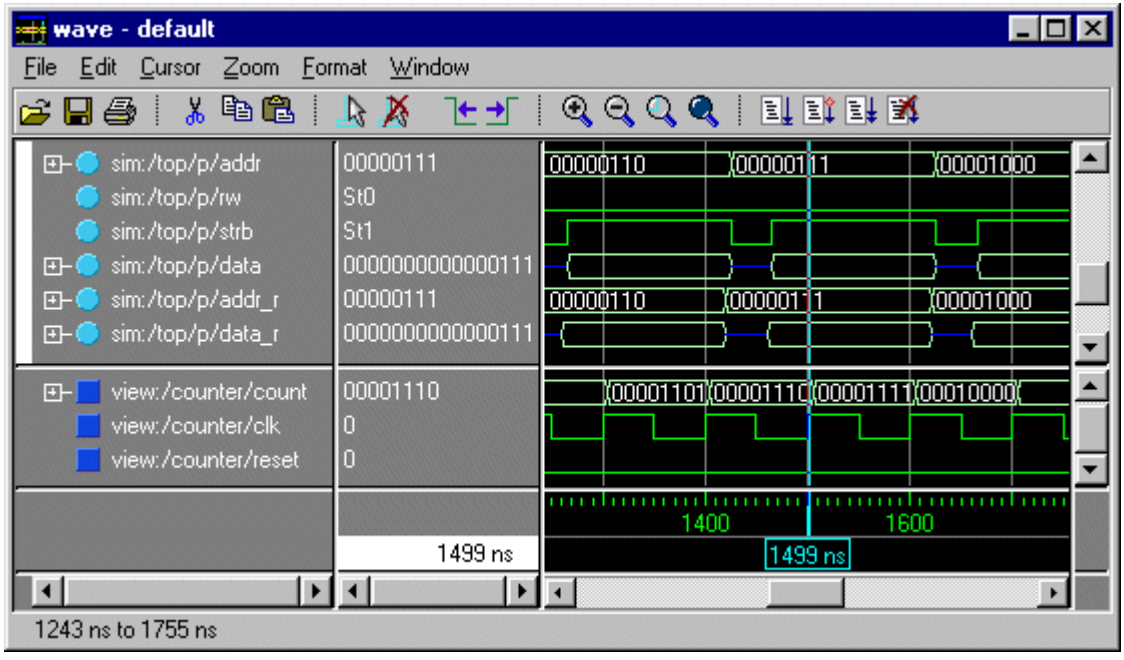

Using dividers

## Wave window panes

The pathnames, values and waveforms window panes of the Wave window display may be split to accommodate signals from one or more datasets. Selecting **File > New Window Pane** creates a space below the selected waveset and makes the new window pane the selected pane. (The selected wave window pane is indicated by a white bar along the left margin of the pane.)

In the illustration below, the Wave window is split, showing the current active simulation with the prefix "sim," and a view-mode simulation, with the prefix "view."

For more information on datasets see *Chapter 6 - Multiple logfiles, datasets and virtuals*.




## HDL items you can view

*VHDL items*

(indicated by a dark blue square)
signals and process variables

*Verilog items*

(indicated by a lighter blue circle)
nets, register variables, and named events

*Virtual items*

(indicated by an orange diamond)
virtual signals, buses, and functions, see
"Virtual Objects (User-defined buses, and more)" (6-100)
for more information

**Note:** Constants, generics, parameters, and memories are not viewable in the List or Wave windows.

The data in the item values windowpane is very similar to the Signals window, except that the values change dynamically whenever a cursor in the waveform windowpane is moved.

At the bottom of the waveform windowpane you can see a time line, tick marks, and a readout of each cursor's position. As you click and drag to move a cursor, the time value at the cursor location is updated at the bottom of the cursor.

You can resize the window panes by clicking on the bar between them and dragging the bar to a new location.

Waveform and signal-name formatting are easily changed via the Format menu (7-171). You can reuse any formatting changes you make by saving a Wave window format file, see "Adding items with a Wave window format file" (7-176).

## The Wave window menu bar

The following menu commands and button options are available from the Wave window menu bar. If you see a dotted line at the top of a drop-down menu, you can select it to create a separate menu window. Many of these commands are also available via a shortcut menu by clicking your right mouse button within the wave window itself.

### File menu

| | |
|---|---|
| Open Dataset | opens a new .wlf dataset file |
| New Divider | inserts divider at current location |
| New Group | allows setup of new group element – a container for other items that can be moved, cut and pasted like other objects |

| | |
|---|---|
| Save Format | saves the current Wave window display and signal preferences to a .do (macro) file; running the .do file will reformat the Wave window to match the display as it appeared when the .do file was created |
| Load Format | run a Wave window format (.do) file previously saved with Save Format |
| Page Setup | allow setup of page for printing; options include: paper size, margins, label width, cursors, color, scaling and orientation |
| Print | sends contents of Wave window to a selected printer; options include:<br>All – prints all signals<br>Current View – prints signals in current view for the time displayed<br>Custom – prints all or current view signals for user-designated time |
| Print Postscript | save or print the waveform display as a Postscript file; options include:<br>All – prints all signals<br>Current View – prints signals in current view for the time displayed<br>Custom – prints all or current view signals for user-designated time |
| New Window Pane | splits the pathname, values and waveform window panes to provide room for a new waveset; gives the option of inserting the new waveset above or below the current, selected, waveset |
| Remove Window Pane | Removes window split and active waveset |
| Refresh Display | clears the Wave window, empties the file cache, and the rebuilds the window from scratch |
| Close | close this copy of the Wave window |

### Edit menu

| | |
|---|---|
| Cut | cut the selected item from the wave name pane; see "Editing and formatting HDL items in the Wave window" (7-178) |
| Copy | copy the selected item and waveform |
| Paste | paste the previously cut or copied item above the currently selected item |
| Delete | delete the selected item and its waveform |
| Select All<br>Unselect All | select, or unselect, all item names in name pane |

| | |
|---|---|
| Combine | combine the selected fields into a user-defined bus |
| Sort | sort the top-level items in the name pane; sort with full path name or viewed name; use ascending, descending or declaration order |
| Find... | find specified item label within the Wave name window |
| Display Properties | set display properties for signal name width, cursor snap distance (in pixels); set dataset prefix to: show all dataset prefixes, show all dataset prefixes if 2 or more, or, show no dataset prefixes; set value justification to left or right margin |
| Signal Properties | set label, height, color, radix, and format for the selected item (use the Format menu selections below to quickly change individual properties) |

### Cursors menu

| | |
|---|---|
| Add Cursor | add a cursor to the center of the waveform window |
| Delete Cursor | delete the selected cursor from the window |
| Goto | choose a cursor to go to from a list of current cursors |

### Zoom menu

| | |
|---|---|
| Zoom <selection> | selection: Full, In, Out, Last, Area with mouse button 1, or Range to change the waveform display range |

### Format menu

| | |
|---|---|
| Radix | set the selected item's radix |
| Format | set the waveform format for the selected item – Literal, Logic, Analog |
| Color | set the color for the selected item from a color palette |
| Height | set the waveform height in pixels for the selected item |

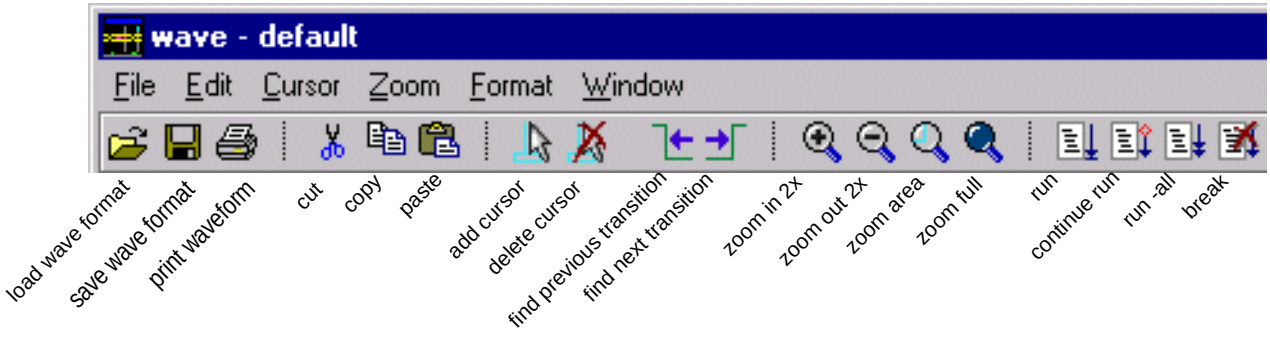
### Window menu

| | |
|---|---|
| Initial Layout | restore all windows to the size and placement of the initial full-screen layout |
| Cascade | cascade all open windows |

| | |
|---|---|
| Tile Horizontally | tile all open windows horizontally |
| Tile Vertically | tile all open windows vertically |
| Icon Children | icon all but the Main window |
| Icon All | icon all windows |
| Deicon All | deicon all windows |
| <window_name> | lists the currently open windows; select a window name to switch to, or show that window if it is hidden; when the source window is available, the source file name is also indicated; open additional windows from the "View menu" (7-116) in the Main window, or use the **view** command (CR-116) |

## Wave window tool bar

The Wave window tool bar gives you quick access to these Model*Sim* commands and functions.




| Wave window tool bar buttons | | |
|---|---|---|
| **Button** | **Menu equivalent** | **Other options** |
| **Load Wave Format** run a Wave window format (DO) file previously saved with Save Format | File > Load Format | none |

| Wave window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Other options** |
| | **Save Wave Format** saves the current Wave window display and signal preferences to a do (macro) file | File > Save Format | none |
| | **Print Waveform** prints a user-selected range of the current Wave window display to a printer or a file | File >Print | none |
| | **Cut** cut the selected signal within the Wave window | Edit > Cut | none |
| | **Copy** copy the selected signal in the signal-name pane | Edit > Copy | none |
| | **Paste** paste the copied signal above another selected signal | Edit > Paste | none |
| | **Add Cursor** add a cursor to the center of the waveform pane | Cursor > Add Cursor | none |
| | **Delete Cursor** delete the selected cursor from the window | Cursor > Delete Cursor | none |

| Wave window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Other options** |
| | **Find Previous Transition** locate the previous signal value change for the selected signal | Edit > Find > Reverse | Keyboard: Shift + Tab |
| | **Find Next Transition** locate the next signal value change for the selected signal | Edit > Find > Forward | Keyboard: Tab |
| | **Zoom in 2x** zoom in by a factor of two from the current view | Zoom > Zoom In | Keyboard: i I or + |
| | **Zoom out 2x** zoom out by a factor of two from current view | Zoom > Zoom Out | Keyboard: o O or - |
| | **Zoom area with mouse button 1** use the cursor to outline a zoom area | Zoom > Zoom Range | Keyboard: r or R |
| | **Zoom Full** zoom out to view the full range of the simulation from time 0 to the current time | Zoom > Zoom Full | Keyboard: f or F |
| | **Run** run the current simulation for the default time length | Main menu: Run > Run <default_length> | see: **run** (CR-86) |

| Wave window tool bar buttons | | | |
|---|---|---|---|
| **Button** | | **Menu equivalent** | **Other options** |
| | **Continue Run**<br>continue the current simulation run | Main menu:<br>Run > Continue | see: **run** (CR-86) |
| | **Run -All**<br>run to current simulation forever, or until it hits a breakpoint or specified break event* | Main menu:<br>Run > Run -All | see: **run** (CR-86), also see "Assertion settings page" (7-213) |
| | **Break**<br>stop the current simulation run | none | none |

## Adding HDL items in the Wave window

Before adding items to the Wave window you may want to set the window display properties (see "Setting Wave window display properties" (7-182)). You can add items to the Wave window in several ways.

### Adding items from the Signals window with drag and drop

You can drag and drop items into the Wave window from the Process, Signals, or Structure window. Select the items in the first window, then drop them into the Wave window. Depending on what you select, all items or any portion of the design may be added.

### Adding items from the Main window command line

To add specific HDL items to the window, enter (separate the item names with a space):

```
add wave <item_name> <item_name>
```

You can add all the items in the current region with this command:

```
add wave *
```

Or add all the items in the design with:

```
add wave -r /*
```

### Adding items with a Wave window format file

To use a Wave window format file you must first save a format file for the design you are simulating.

- add the items you want in the Wave window with any other method shown above
- edit and format the items, see "Editing and formatting HDL items in the Wave window" (7-178) to create the view you want
- save the format to a file with the Wave window menu selection: **File > Save Format**

To use the format file, start with a blank Wave window and run the DO file in one of two ways:

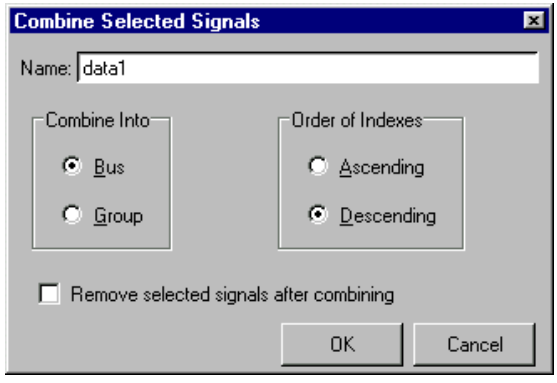
- use the do command on the command line:
  `do <my_wave_format>`
- select **File > Load Format** from the Wave window menu bar

Use **Edit > Select All** and **Edit > Delete** to remove the items from the current Wave window, use the **delete** command (CR-40) with the **wave** option, or create a new, blank Wave window with the **View > New > Wave** selection from the Main window (7-113).

**Note:** Wave window format files are design-specific; use them only with the design you were simulating when they were created.

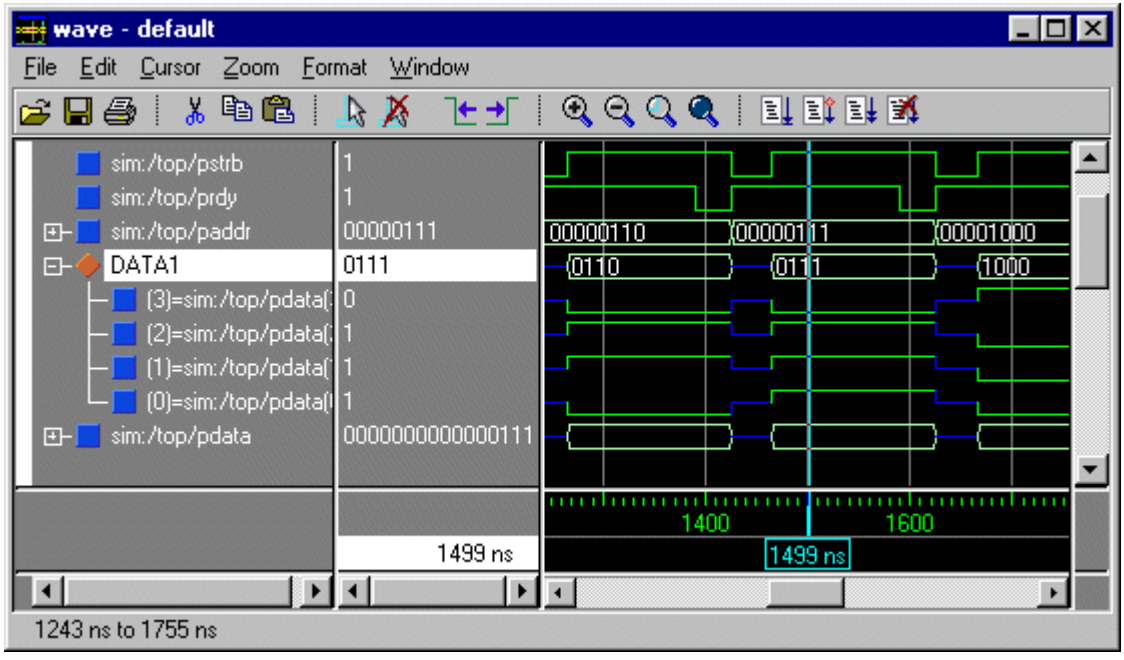
## Combining and grouping items in the Wave window

The Wave window allows you to combine signals into busses or groups. Use the **Edit > Combine** menu selections to call up the Combine Selected Signals Dialog box.

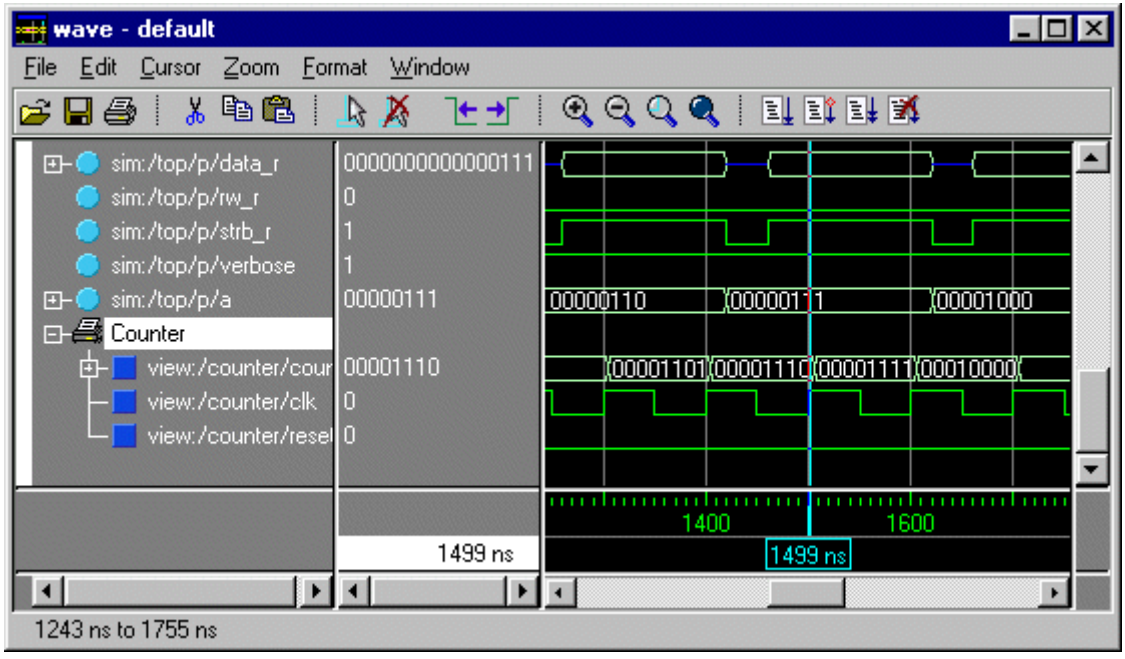



A bus is a collection of signals concatenated in a specific order to create a new virtual signal with a specific value.

In the illustration below, four data signals have been combined to form a new bus called DATA1. Notice, the new bus has a value that is made up of the values of its component signals arranged in a specific order. Virtual objects are indicated by an orange diamond.




A group is simply a container for any number of signals. It has no value, and the signals contained within it may be arranged in any order. In the illustration below, the signals counter/count, counter/clk, and counter/reset have been combined in a group called Counter. Notice that the Counter group has no value associated with it. The counter, clk and reset signals may be arranged in any order.




### Other virtual items in the Wave window

See "Virtual Objects (User-defined buses, and more)" (6-100)
for information about other virtual item viewable in the Wave window.

## Editing and formatting HDL items in the Wave window

Once you have the HDL items you want in the Wave window, you can edit and format the list in the name/value pane to create the view you find most useful. (See also, "Setting Wave window display properties" (7-182).)

To edit an item:

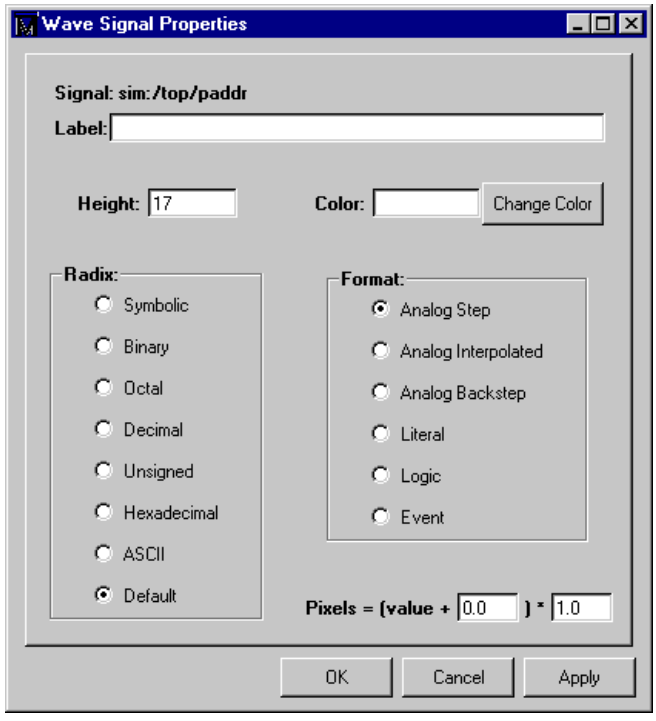
Select the item's label in the left name/value windowpane or its waveform in the right windowpane. Move, copy or remove the item by selecting commands from the Wave window Edit menu (7-170) menu.

You can also **click+drag** to move items within the name/value windowpane:

- to select several contiguous items:
  click+drag to select additional items above or below the original selection
- to select several items randomly:
  control+click to add or subtract from the selected group
- to move the selected items:
  re-click and hold on one of the selected items, then drag to the new location

To format an item:

Select the item's label in the left pathname pane or its waveform in the waveform pane, then use the **Edit > Signal Properties** menu selection. The resulting Wave Signal Properties dialog box allows you to set the item's height, color, format, range, and radix.




The **Wave Signal Properties** dialog box includes these options:

- **Signal**
Indicates the name of the currently selected signal.

- **Label**
Allows you to specify a new label (in the pathname pane) for the selected item.

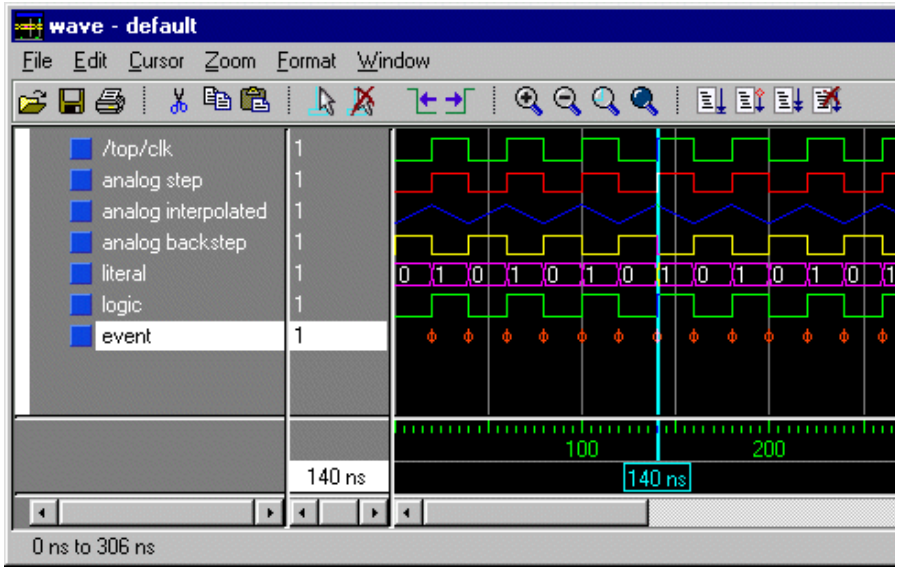
- **Height**
Allows you to specify the height (in pixels) of the waveform.

- **Color**
Lets you override the default color of a waveform by selecting a new color from the color palette, or by entering an X-Windows color name.

- **Pixels = (value + <offset>) * <scale factor>**
This choice works with analog items only and allows you to decide on the scale of the item as it is seen on the display. Value is the value of the signal at a given time, <offset> is the number of pixels offset from zero. The <scale factor> reduces (if less than 1) or increases (if greater than 1) the number of pixels displayed.

- **Radix**
The explicit choices are Symbolic, Binary, Octal, Decimal, Unsigned, Hexadecimal, and ASCII. If you select Default the signal's radix changes whenever the default is changed using the **radix** command (CR-81). Item values are not translated if you select Symbolic.

- **Format: Analog [Step | Interpolated | Backstep]**




All signals in this illustration are the same */top/clk* signal. Starting with "analog step", the */top/clk* signal has been relabeled to illustrate each different wave formats.

*Analog Step*
Displays a waveform in step style.

*Analog Interpolated*
Displays the waveform in interpolated style.

*Analog Backstep*
Displays the waveform in backstep style. Often used for power calculations.

Only the following types are supported in Analog format:

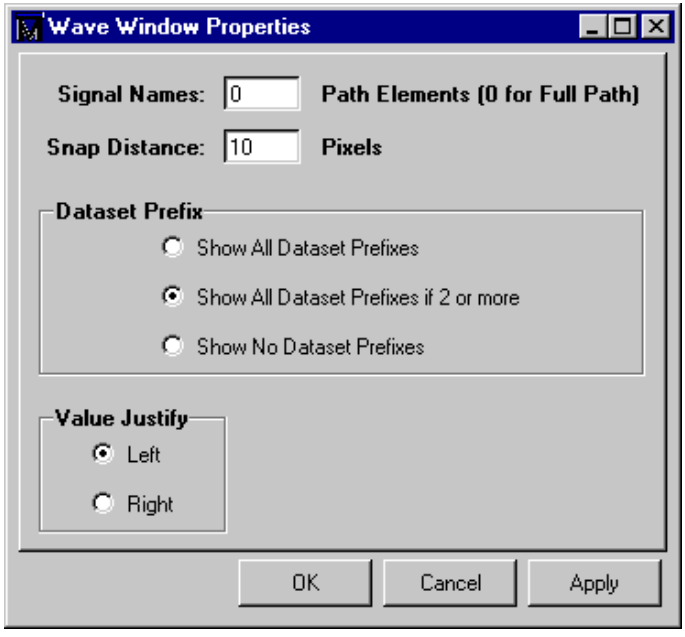
VHDL types:
All vectors - std logic vectors, bit vectors, and vectors derived from these types
Scaler integers
Scaler reals
Scaler time

Verilog types:
All vectors
Scaler real
Scalar integers

Wave height and offset can be set from the Wave window menu Prop > Signal props by adjusting the parameters to the Pixels equation.

- **Format: Literal**
  Displays the waveform as a box containing the item value (if the value fits the space available). This is the only format that can be used to list a record.
- **Format: Logic**
  Displays values as 0, 1, X, Z, H, L, U, or -.
- **Format: Event**
  Marks each transition during the simulation run.

## Setting Wave window display properties




You can define the display properties of the pathname and values window panes by selecting **Edit > Display Properties** in the Wave window.

The **Wave Window Properties** dialog box includes the following options:

- **Signal Names**
  This selection allows you to display the full pathname of each signal (i.e.: *sim:/top/clk)*, or its leaf element only (i.e.: *sim:clk*). The default is Full Path.
- **Snap Distance**
  Specifies the distance the cursor needs to be placed from an item edge to jump to that edge (a 0 specification turns off the snap). The value displayed in the item value windowpane is updated to reflect the snap.
- **Dataset Prefix**
  With this selection you can determine how signals from different datasets are displayed.

  *Show All Dataset Prefixes*
  All dataset prefixes will be displayed along with the dataset prefix of the current simulation ("sim").

*Show All Dataset Prefixes if 2 or more*
Displays all dataset prefixes if 2 or more datasets are displayed. "sim" is the default prefix for the current simulation.

*Show No Dataset Prefixes*
No dataset prefixes will be display. This selection is useful if you are only running a single simulation.

- **Value Justify**
  Specifies whether the signal values will be justified to the left margin or the right margin in the values window pane.

## Sorting a group of HDL items

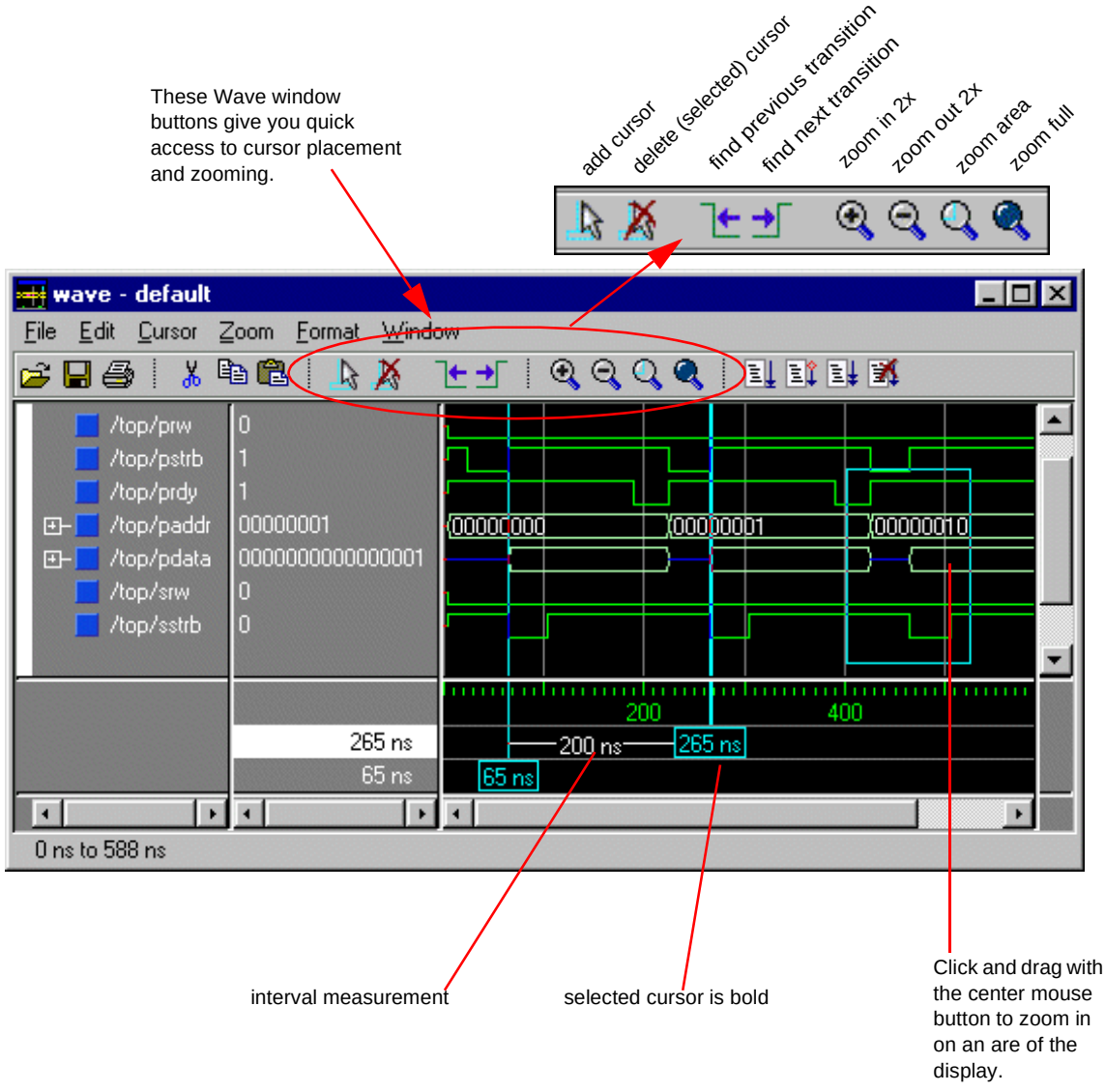
Use the **Edit** > **Sort** menu selection to sort the items in the name/value pane.

## Finding items by name or value in the Wave window

The Find dialog box allows you to search for text strings in the Wave window. From the Wave window select **Edit** > **Find** to bring up the Find dialog box.

Choose either the Name or Value field to search from the drop-down menu, and enter the value to search for in the Find field. **Find** the item by searching **Forward** (down) or **Reverse** (up) through the Wave window display.

## Using time cursors in the Wave window




When the Wave window is first drawn, there is one cursor located at time zero. Clicking anywhere in the waveform display brings that cursor to the mouse location.You can add additional cursors to the waveform pane with the **Cursor** > **Add Cursor** menu selection (or the Add Cursor button shown below). The

selected cursor is drawn as a bold solid line; all other cursors are drawn with thin solid lines. Remove cursors by selecting them and choosing using the **Cursor > Delete Cursor** menu selection (or the Delete Cursor button shown below).

| | |
|---|---|
| **Add Cursor** add a cursor to the center of the waveform window | **Delete Cursor** delete the selected cursor from the window |

### Finding a cursor

The cursor value (on the **Goto** list) corresponds to the simulation time of that cursor. Choose a specific cursor view with **Cursor > Goto** menu selection.

### Making cursor measurements

Each cursor is displayed with a time box showing the precise simulation time at the bottom. When you have more than one cursor, each time box appears in a separate track at the bottom of the display. VSIM also adds a delta measurement showing the time difference between the two cursor positions.

If you click in the waveform display, the cursor closest to the mouse position is selected and then moved to the mouse position. Another way to position multiple cursors is to use the mouse in the time box tracks at the bottom of the display. Clicking anywhere in a track selects that cursor and brings it to the mouse position.

The cursors are designed to snap to the closest wave edge to the left on the waveform that the mouse pointer is positioned over. You can control the snap distance from "Wave category" in the dialog box available from the Wave window **Prop > Display Props** menu selection.

You can position a cursor without snapping by dragging in the area below the waveforms.

You can also move cursors to the next transition of a signal with these toolbar buttons:

| | |
|---|---|
| **Find Previous Transition** locate the previous signal value change for the selected signal | **Find Next Transition** locate the next signal value change for the selected signal |

## Zooming - changing the waveform display range

Zooming lets you change the simulation range in the windowpane display. You can zoom with either the **Zoom** menu, toolbar buttons, mouse, keyboard, or VSIM commands.

### Using the Zoom menu

You can use the Wave window menu bar, or call up the **Zoom** menu by clicking the right mouse button (of a three-button mouse) in the right windowpane.

**Note:** The right mouse button of a two-button mouse will not open the **Zoom** menu. It will, however, allow you to create a zoom area by dragging left to right while holding down the button.

The Zoom menu options include:

- **Zoom Full**
  Redraws the display to show the entire simulation from time 0 to the current simulation time.
- **Zoom In**
  Zooms in by a factor of two, increasing the resolution and decreasing the visible range horizontally, cropping the view on the right. The starting time is held static.
- **Zoom Out**
  Zooms out by a factor of two, decreasing the resolution and increasing the visible range horizontally, extending the view on the right. The starting time is held static.
- **Zoom Last**
  Restores the display to where it was before the last zoom operation.
- **Zoom Area with Mouse Button 1**
  Use mouse button 1 to create a zoom area. Position the mouse cursor to the left side of the desired zoom interval, press mouse button 1 and drag to the right. Release when the box has expanded to the right side of the desired zoom interval.
- **Zoom Range**
  Brings up a dialog box that allows you to enter the beginning and ending times for a range of time units to be displayed.

### Zooming with the toolbar buttons

These zoom buttons are available on the toolbar:

| | |
|---|---|
| **Zoom in 2x** zoom in by a factor of two from the current view | **Zoom area** use the cursor to outline a zoom area |
| **Zoom out 2x** zoom out by a factor of two from current view | **Zoom Full** zoom out to view the full range of the simulation from time 0 to the current time |

### Zooming with the mouse

To zoom with the mouse, position the mouse cursor to the left side of the desired zoom interval, press the middle mouse button (three-button mouse), or right button (two-button mouse), and while continuing to press, drag to the right and then release at the right side of the desired zoom interval.

### Zooming keyboard shortcuts

See "Wave window keyboard shortcuts" (7-187) for a complete list of Wave window keyboard shortcuts.

## Wave window keyboard shortcuts

Using the following keys when the mouse cursor is within the Wave window will cause the indicated actions:

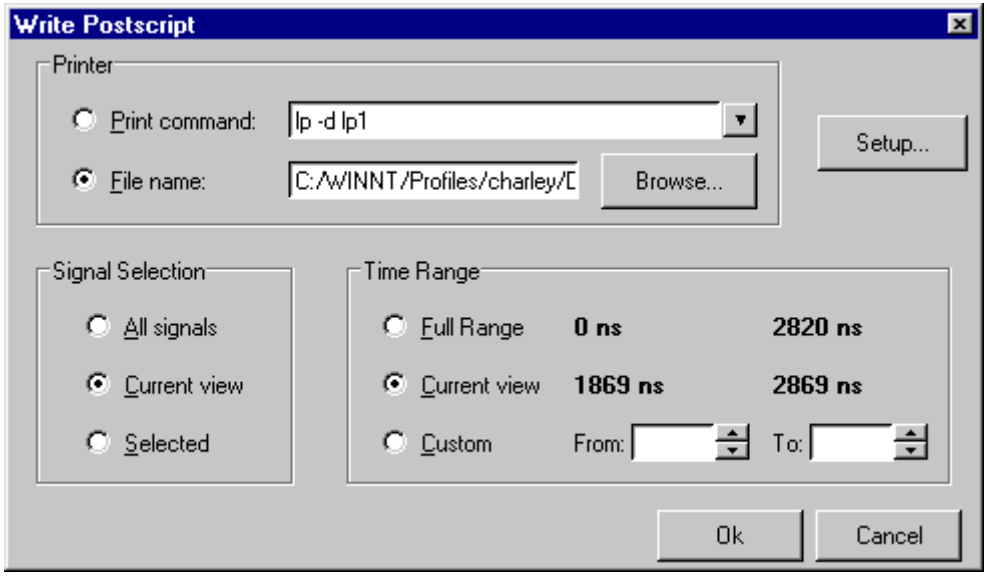
| Key | Action |
|---|---|
| i I or + | zoom in |
| o O or - | zoom out |
| f or F | zoom full |
| l or L | zoom last |
| r or R | zoom range |

| Key | Action |
|---|---|
| <arrow up> | scroll waveform display up |
| <arrow down> | scroll waveform display down |
| <arrow left> | scroll waveform display left |
| <arrow right> | scroll waveform display right |
| <page up> | scroll waveform display up by page |
| <page down> | scroll waveform display down by page |
| <tab> | searches forward (right) to the next transition on the selected signal - finds the next edge |
| <shift-tab> | searches backward (left) to the previous transition on the selected signal - finds the previous edge |
| <control-f> | opens the find dialog box; search within the specified field in the wave-name pane for text strings |

## Saving waveforms

### Saving a .eps file

Use the **File > Print Postscript** menu selection in the Wave window to save the waveform as a .eps file. Printing and writing preferences are controlled by the dialog box shown below.




The **Write Postscript** dialog box includes these options:

*Printer*

- **File name**
  enter a filename for the encapsulated Postscript (.eps) file to be created; or browse to a previously created .eps file and use that filename.

*Signal Selection*

- **All signals**
  prints all signals
- **Current View**
  prints signals in current view
- **Selected**
  prints all selected signals

*Time Range*

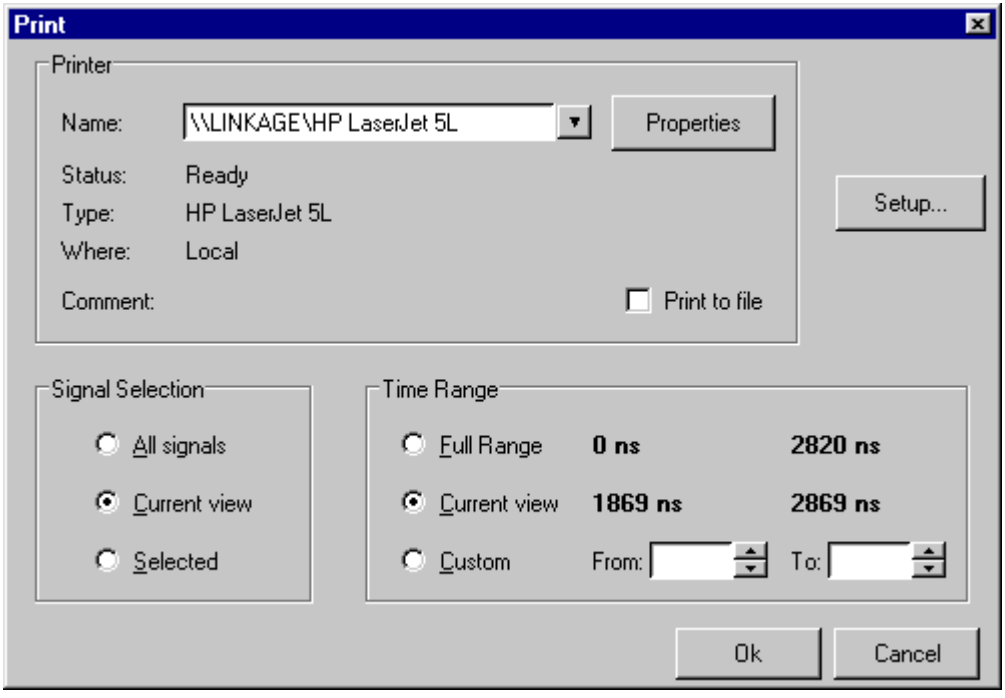
- **Full Range**
  prints all specified signals in the full simulation range
- **Current view**
  prints specified signals for the viewable time range
- **Custom**
  prints the specified signals for user-designated **From** and **To** time

*Setup button*

See "Printer Page Setup" (7-193)

### Printing on Windows platforms

Use the **File > Print** menu selection in the Wave window to print all or part of the waveform in the current Wave window, or save the waveform as a printer file (a Postscript file for Postscript printers). Printing and writing preferences are controlled by the dialog box shown below.




*Printer*

- **Name**
  Choose the printer from the drop-down menu. Set printer properties with the *Properties* button.
- **Status**
  Indicates the availability of the selected printer.
- **Type**
  Printer driver name for the selected printer. The driver determines what type of file is output if "Print to file" is selected.

- **Where**
  The printer port for the selected printer.
- **Comment**
  The printer comment from the printer properties dialog box.
- **Print to file**
  Make this selection to print the waveform to a file instead of a printer. The printer driver determines what type of file is created. Postscript printers create a Postscript (.ps) file, non-Postscript printers print a .prn or printer control language file. To create an encapsulated Postscript file (.eps) use the File > Print Postscript menu selection.

*Signal Selection*

- **All signals**
  prints all signals
- **Current View**
  prints signals in current view
- **Selected**
  prints all selected signals

*Time Range*

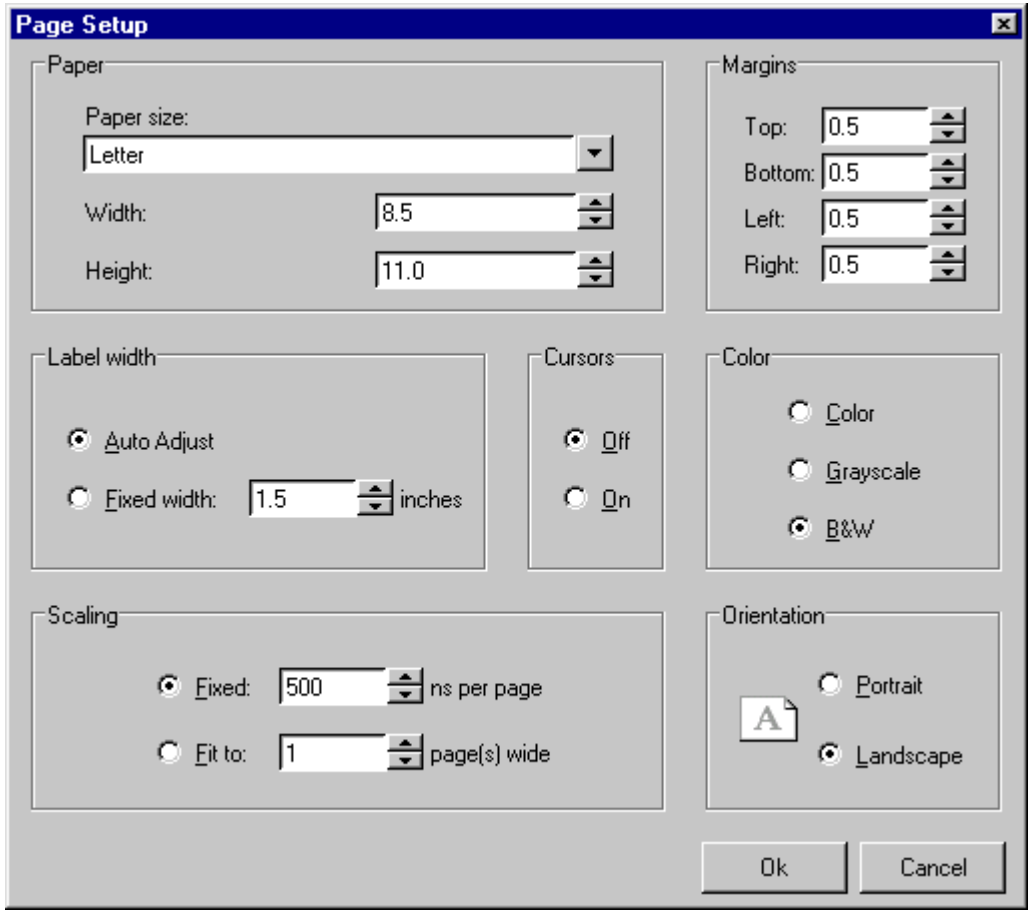
- **Full Range**
  prints all specified signals in the full simulation range
- **Current view**
  prints specified signals for the viewable time range
- **Custom**
  prints the specified signals for user-designated **From** and **To** time

*Setup button*

See "Printer Page Setup" (7-193)

### Printer Page Setup

Clicking the Setup button in the Write Postscript or Print dialog box allows you to define the following options (this is the same dialog that opens with File > Page setup menu selection).




- **Paper Size**
  select your output page size from a number of options; also choose the paper width and height
- **Margins**
  specify the margin in inches or centimeters (units are controlled by the inches/cm selection); changing the **Margin** will change the **Scale** and **Page** specifications

- **Label width**
  specify Auto Adjust to accommodate any length label, or set a fixed label width
- **Cursors**
  turn printing of cursors on or off
- **Color**
  select full color printing, grayscale or black and white
- **Scaling**
  specify a **Fixed** output time width in nanoseconds per page – the number of pages output is automatically computed; or, select **Fit to** to define the number of pages to be output based on the paper size and time settings; if set, the time-width per page is automatically computed
- **Orientation**
  select the output page orientation, **Portrait** or **Landscape**

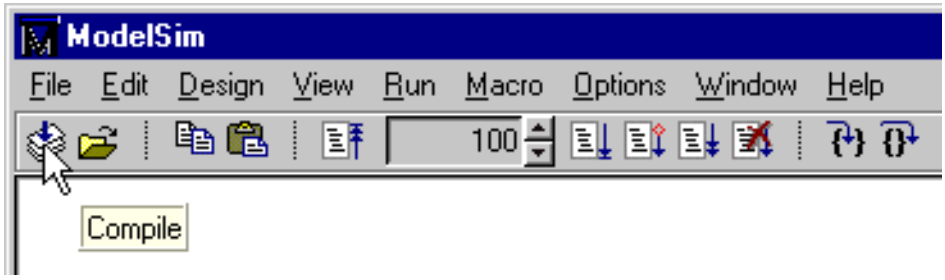
### The vsim.ps include file

The Postscript file that VSIM creates (if you print to a file) includes a file named *vsim.ps* that is shipped with Model*Sim*. The file is "included" in the Postscript output from the waveform display; it sets the fonts, spacing, and print header and footer (the font and spacing is based on those used in the Wave window). If you want to change any of the Postscript defaults, you can do it by making changes in this file. Note that you should copy the file before making your changes so that you can save the original.

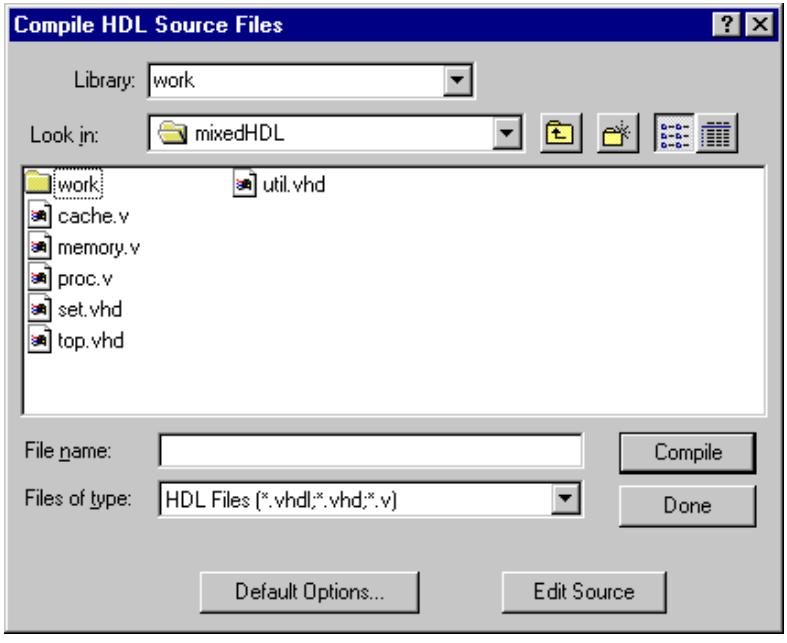
```
% Copyright 1993 Model Technology Incorporated.
% All rights reserved.
% A(#)vsim.ps 1.2 13 Mar 1994
%
% This file is 'included' in the postscript output from the
% waveform display.
%
% pick the fonts
/fontheight 10 def
/mainfont {/Helvetica-Narrow findfont fontheight scalefont setfont} def
/smallfont {/Helvetica-Narrow findfont fontheight 3 sub scalefont setfont} def
mainfont
3 10 div setlinewidth
/signal_spacing fontheight 9 add def
/one_ht fontheight 2 sub def
/z_ht one_ht 2 div def
/ramp 2 def
/hz_tick_len 4 def
...
```

## Compiling with the graphic interface

To compile either VHDL or Verilog designs, select the **Compile** button on the Main window toolbar.




The **Compile HDL Source Files** dialog box opens as shown below.




From the **Compile HDL Source Files** dialog box you can:

- select source files to compile in any language combination
- specify the target library for the compiled design units
- select among the compiler options for either VHDL or Verilog

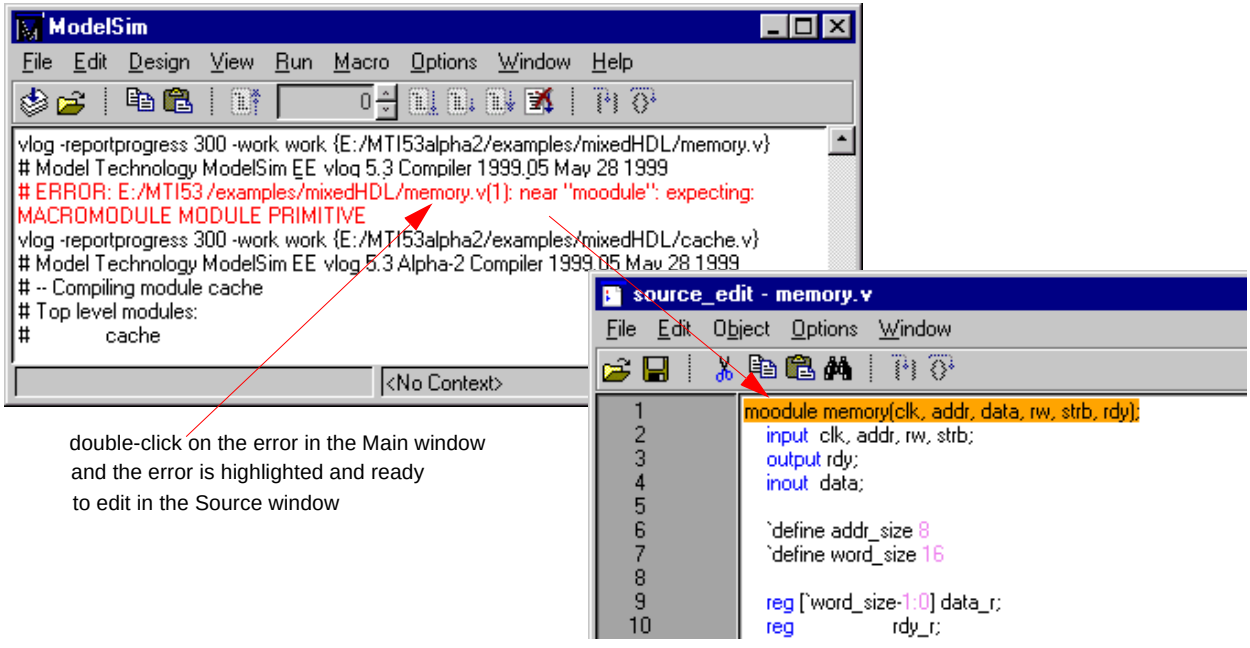
Select the **Default Options** button to change the compiler options, see "Setting default compile options" (7-196) for details. The same Compiler Options dialog box can also be accessed with the **Options > Compile** Main window menu selection.

Select the **Edit Source** button to view or edit a source file via the Compile dialog box. See "Source window" (7-152), and "Mouse and keyboard shortcuts in the

Transcript and Source windows" (7-121) for additional source file editing information.

## Locating source errors during compilation

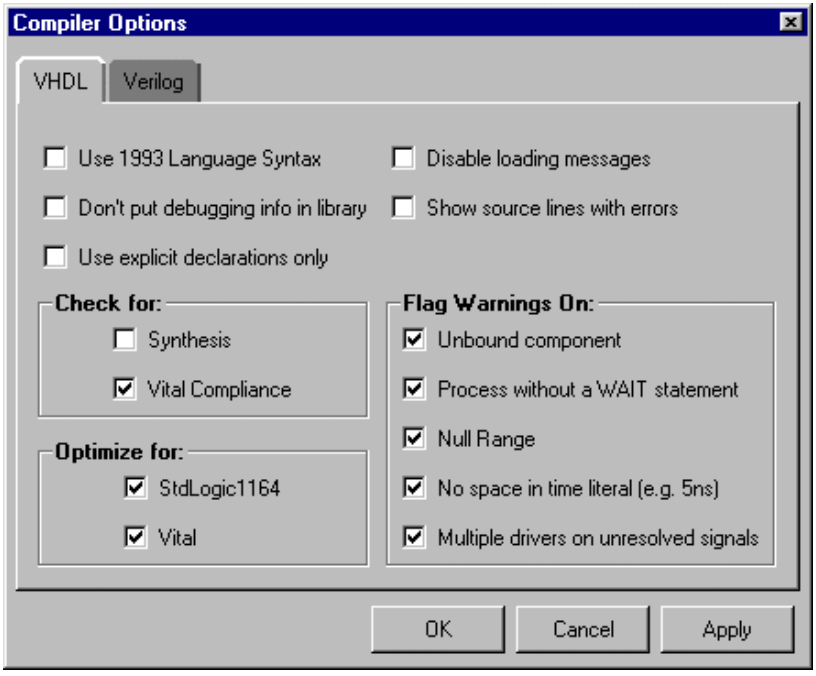
If a compiler error occurs during compilation, a red error message is printed in the Main transcript. Double-click on the error message to open the source file in an editable Source window with the error highlighted.


ModelSim
File Edit Design View Run Macro Options Window Help
vlog -reportprogress 300 -work work {E:/MTI53alpha2/examples/mixedHDL/memory.v}
# Model Technology ModelSim EE vlog 5.3 Compiler 1999.05 May 28 1999
# ERROR: E:/MTI53/examples/mixedHDL/memory.v(1): near "moodule": expecting:
MACROMODULE MODULE PRIMITIVE
vlog -reportprogress 300 -work work {E:/MTI53alpha2/examples/mixedHDL/cache.v}
# Model Technology ModelSim EE vlog 5.3 Alpha-2 Compiler 1999.05 May 28 1999
# -- Compiling module cache
# Top level modules:
# cache
<No Context>
source_edit - memory.v
File Edit Object Options Window
moodule memory(clk, addr, data, rw, strb, rdy);
input clk, addr, rw, strb;
output rdy;
inout data;
`define addr_size 8
`define word_size 16
reg [`word_size-1:0] data_r;
reg rdy_r;
double-click on the error in the Main window
and the error is highlighted and ready
to edit in the Source window

## Setting default compile options

Use the **Options > Compile** menu selection to bring up the **Compile Options** dialog box shown below. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page of the dialog box are detailed below. Changes made in the **Compile Options** dialog box become the default for all future simulations.

VHDL compiler options page




- **Use 1993 language syntax**
  Specifies the use of VHDL93 during compilation. The 1987 standard is the default. Same as the **-93** switch for the **vcom** command (CR-106). Edit the VHDL93 (A-260) in the *modelsim.ini* to set a permanent default.
- **Don't put debugging info in library**
  Models compiled with this option do not use any of the Model*Sim* debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you're done debugging. Same as the **-nodebug** switch for the **vcom** command (CR-106). See "Source code security and -nodebug" (C-292) for more details. Edit the NoDebug (A-261) in the *modelsim.ini* to set a permanent default.
- **Use explicit declarations only**
  Used to ignore an error in packages supplied by some other EDA vendors; directs the compiler to resolve ambiguous function overloading in favor of the

explicit function definition. Same as the **-explicit** switch for the **vcom** command (CR-106). Edit the Explicit (A-261) in the *modelsim.ini* to set a permanent default.

Although it is not intuitively obvious, the = operator is overloaded in the **std_logic_1164** package. All enumeration data types in VHDL get an "implicit" definition for the = operator. So while there is no explicit = operator, there is an implicit one. This implicit declaration can be hidden by an explicit declaration of = in the same package (LRM Section 10.3). However, if another version of the = operator is declared in a different package than that containing the enumeration declaration, and both operators become visible through **use** clauses, neither can be used without explicit naming, i.e.,

```
ARITHMETIC."="(left, right)
```

This option allows the explicit = operator to hide the implicit one.

- **Disable loading messages**
  Disables loading messages in the Transcript window. Same as the **-quiet** switch for the **vcom** command (CR-106). Edit the Quiet (A-261) in the *modelsim.ini* to set a permanent default.
- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vcom** command (CR-106). Edit the Show_source (A-260) in the *modelsim.ini* to set a permanent default.

**Flag Warnings on:**

- **Unbound Component**
  Flags any component instantiation in the VHDL source code that has no matching entity in a library that is referenced in the source code, either directly or indirectly. Edit the Show_Warning1 (A-260) in the *modelsim.ini* to set a permanent default.
- **Process without a wait statement**
  Flags any process that does not contain a wait statement or a sensitivity list. Edit the Show_Warning2 (A-260) in the *modelsim.ini* to set a permanent default.
- **Null Range**
  Flags any null range, such as 0 down to 4. Edit the Show_Warning3 (A-260) in the *modelsim.ini* to set a permanent default.
- **No space in time literal (e.g. 5ns)**
  Flags any time literal that is missing a space between the number and the time unit. Edit the Show_Warning4 (A-260) in the *modelsim.ini* to set a permanent default.

- **Multiple drivers on unresolved signal**
  Flags any unresolved signals that have multiple drivers. Edit the Show_Warning5 (A-260) in the *modelsim.ini* to set a permanent default.

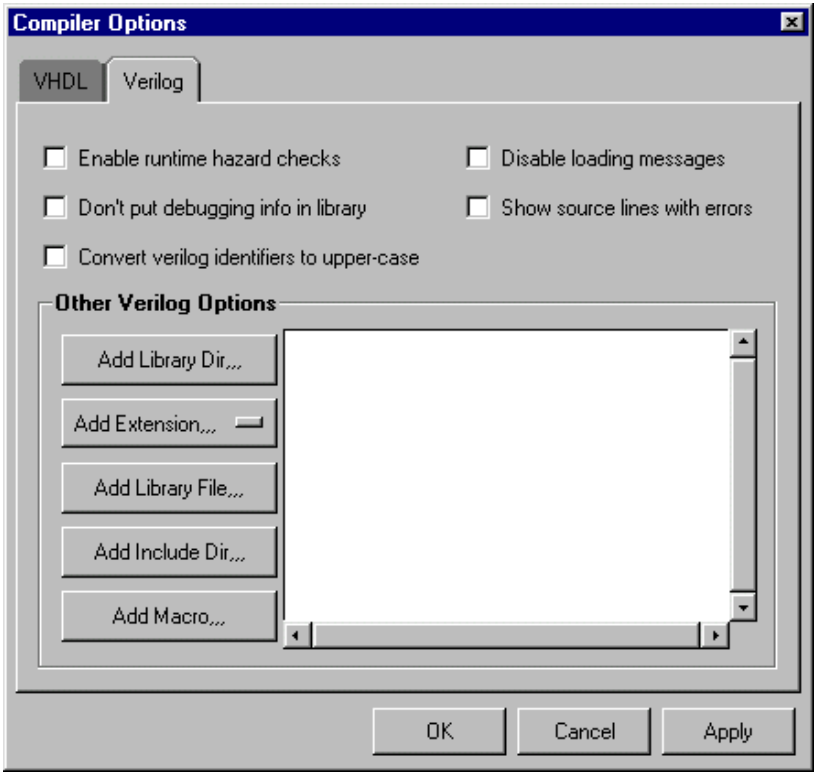
**Check for:**

- **Synthesis**
  Turns on limited synthesis-rule compliance checking. Edit the CheckSynthesis (A-261) in the *modelsim.ini* to set a permanent default.
- **Vital Compliance**
  Toggle Vital compliance checking. Edit the NoVitalCheck (A-261) in the *modelsim.ini* to set a permanent default.

**Optimize for:**

- **std_logic_1164**
  Causes the compiler to perform special optimizations for speeding up simulation when the multi-value logic package std_logic_1164 is used. Unless you have modified the std_logic_1164 package, this option should always be checked. Edit the Optimize_1164 (A-260) in the *modelsim.ini* to set a permanent default.
- **Vital**
  Toggle acceleration of the Vital packages. Edit the NoVital (A-261) in the *modelsim.ini* to set a permanent default.

Verilog compiler options page




- **Enable run-time hazard checks**
  Enables the run-time hazard checking code. Same as the **-hazards** switch for the **vlog** command (CR-141). Edit the Hazard (A-261) in the *modelsim.ini* to set a permanent default.
- **Don't put debugging info in library**
  Models compiled with this option do not use any of the Model*Sim* debugging features. Consequently, your user will not be able to see into the model. This also means that you cannot set breakpoints or single step within this code. Don't compile with this option until you're done debugging. Same as the **-nodebug** switch for the **vlog** command (CR-141). See "Source code security and -nodebug" (C-292) for more details. Edit the NoDebug (A-261) in the *modelsim.ini* to set a permanent default.

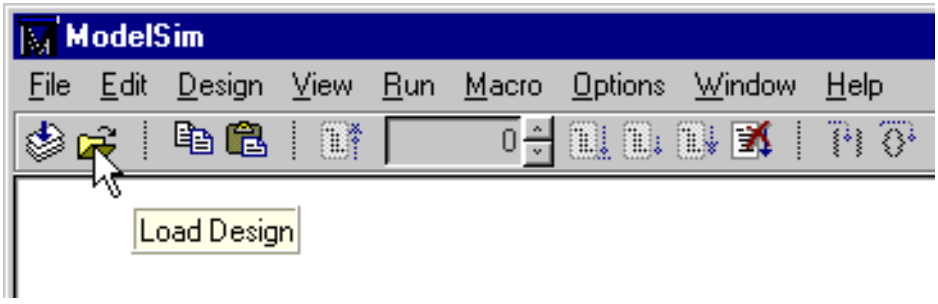
- **Convert Verilog identifiers to upper-case**
  Converts regular Verilog identifiers to uppercase. Allows case insensitivity for module names. Same as the **-u** switch for the **vlog** command (CR-141). Edit the UpCase (A-262) in the *modelsim.ini* to set a permanent default.
- **Disable loading messages**
  Disables loading messages in the Transcript window. Same as the **-quiet** switch for the **vlog** command (CR-141). Edit the Quiet (A-261) in the *modelsim.ini* to set a permanent default.
- **Show source lines with errors**
  Causes the compiler to display the relevant lines of code in the transcript. Same as the **-source** switch for the **vlog** command (CR-141). Edit the Show_source (A-260) in the *modelsim.ini* to set a permanent default.

**Other Verilog Options:**

- **Add Library Dir**
  Specifies the Verilog source library directory to search for undefined modules. Same as the **-y <library_directory>** switch for the **vlog** command (CR-141).
- **Add Extension**
  Specifies the suffix of files in library directory. Multiple suffixes may be used. Same as the **+libext+<suffix>** switch for the **vlog** command (CR-141).
- **Add Library File**
  Specifies the Verilog source library file to search for undefined modules. Same as the **-v <library_file>** switch for the **vlog** command (CR-141).
- **Add Include Dir**
  Search specified directory for files included with the ‘i**nclude filename** compiler directive. Same as the **+incdir+<directory>** switch for the **vlog** command (CR-141).
- **Add Macro**
  Define a macro to execute during compilation. Same as compiler directive: ‘define macro_name macro_text. Also the same as the +define+<macro_name> [ =<macro_text> ] switch for the **vlog** command (CR-141).

## Simulating with the graphic interface

The **Load Design** dialog box is activated when the Load Design button is selected from the Main window toolbar.




Four pages - **Design**, **VHDL**, **Verilog**, and **SDF** - allow you to select various simulation options.

You can switch between pages to modify settings, then begin simulation by selecting the **Load** button. If you select **Cancel**, all selections remain unchanged and you are returned to the Main VSIM window; the **Exit** button (only active before simulation) closes Model*Sim*. The **Save Settings** button allows you to save the preferences on all pages to a do (macro) file.

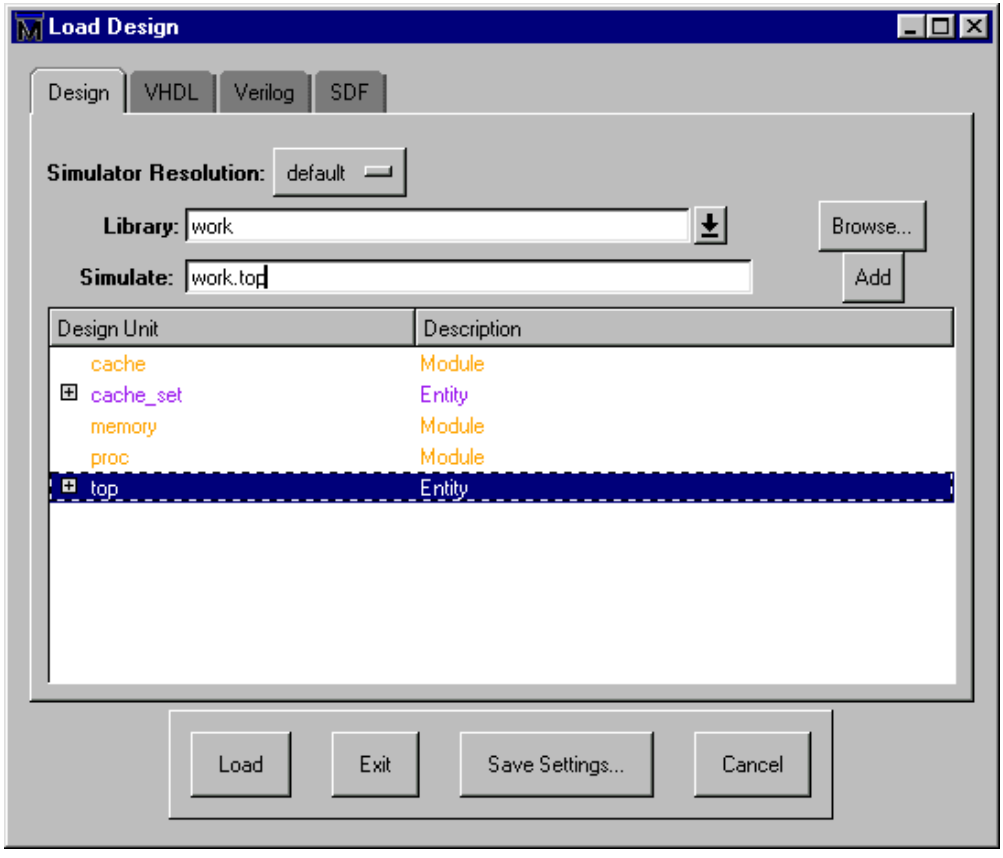
### Compile before you simulate

To begin simulation you must have compiled design units located in a design library, see "Creating a design library" (4-44).

### VSIM command options

Options that correspond to **vsim** (CR-148) commands are noted within parentheses in the text below, i.e., **Simulator Resolution** (-t [<multiplier>]<time_unit>).

## Design selection page




**Note:** The Exit button closes the Load Design dialog box and quits Model*Sim*.

The **Design** page includes these options:

- **Simulator Resolution**
  (-time [<multiplier>]<time_unit>)
  The drop-down menu sets the simulator time units (original default is ns).
- **Library**
  Specifies a library for viewing in the **Design Unit** list box. You can use the drop-down list (click the arrow) to select a "mapped" library or you can type in

a library name. You can also use the **Browse** button to locate a library among your directories. Make certain your selection is a valid Model*Sim* library - it must include an *_info* file and must have been created from Model*Sim*'s **vlib** command (CR-140). Once the library is selected you can view its design units within the **Design Unit** list box.

- **Simulate** (<configuration> | <module> | <entity> [(<architecture>)])
  Specifies the design unit(s) to simulate. You can simulate several Verilog top-level modules or a VHDL top-level design unit in one of three ways:

  **1.** Type a design unit name (configuration, module, or entity) into the field, separate additional names with a space. Specify library/design units with the following syntax:

  ```
  [<library_name>.]<design_unit>
  ```

  **2.** Click on a name in the **Design Unit** list below and click the **Add** button.
  **3.** Leave this field blank and click on a name in the **Design Unit** list (single unit only).

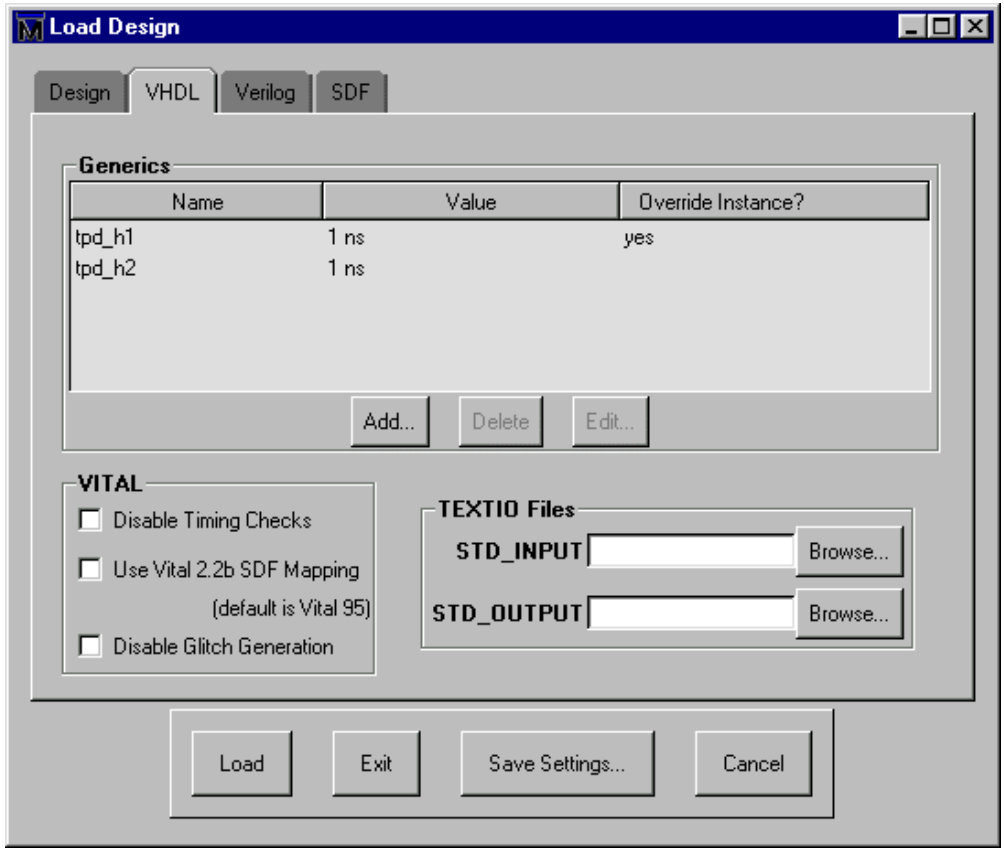
- **Design Unit/Description**
  This hierarchal list allows you to select one top-level entity or configuration to be simulated. All entities and configurations that exist in the specified library are displayed in the list box. Architectures may be viewed by selecting the "+" box before any name.

Simulator time units may be expressed as any of the following:

| Simulation time units | |
|---|---|
| 1fs, 10fs, or 100fs | femtoseconds |
| 1ps, 10ps, or 100ps | picoseconds |
| 1ns, 10ns, or 100ns | nanoseconds |
| 1us, 10us, or 100us | microseconds |
| 1ms, 10ms, or 100ms | milliseconds |
| 1sec, 10sec, or 100sec | seconds |

See also, "Selecting the time resolution" (4-46).

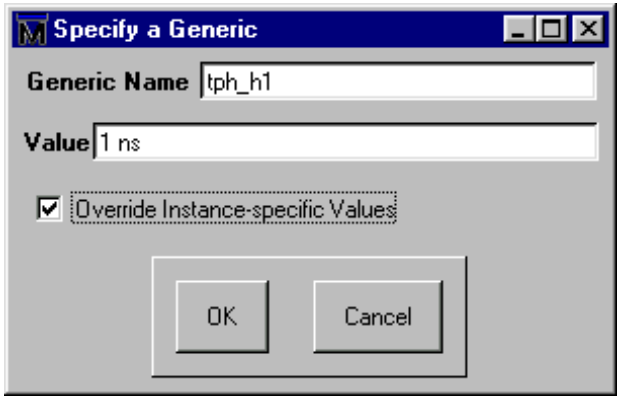
## VHDL settings page




The **VHDL** page includes these options:

**Generics**

The **Add** button opens a dialog box that allows you to specify the value of generics within the current simulation; generics are then added to the **Generics** list. You may also select a generic on the listing to **Delete** or **Edit** (opens the dialog box below).




From **Specify a Generic** dialog box you can set the following options.

- **Generic Name** (-g <Name=Value>)
The name of the generic parameter. You can make a selection from the drop-down menu or type it in as it appears in the VHDL source (case is ignored).

- **Value**
Specifies a value for all generics in the design with the given name (above) that have not received explicit values in generic maps (such as top-level generics and generics that would otherwise receive their default value). Value is an appropriate value for the declared data type of the generic parameter. No spaces are allowed in the specification (except within quotes) when specifying a string value.

- **Override Instance - specific Values** (-G <Name=Value>)
Select to override generics that received explicit values in generic maps. The name and value are specified as above. The use of this switch is indicated in the **Override Instance** column of the **Generics** list.

The **OK** button adds the generic to the **Generics** listing; **Cancel** dismisses the dialog box without changes.

**VITAL**

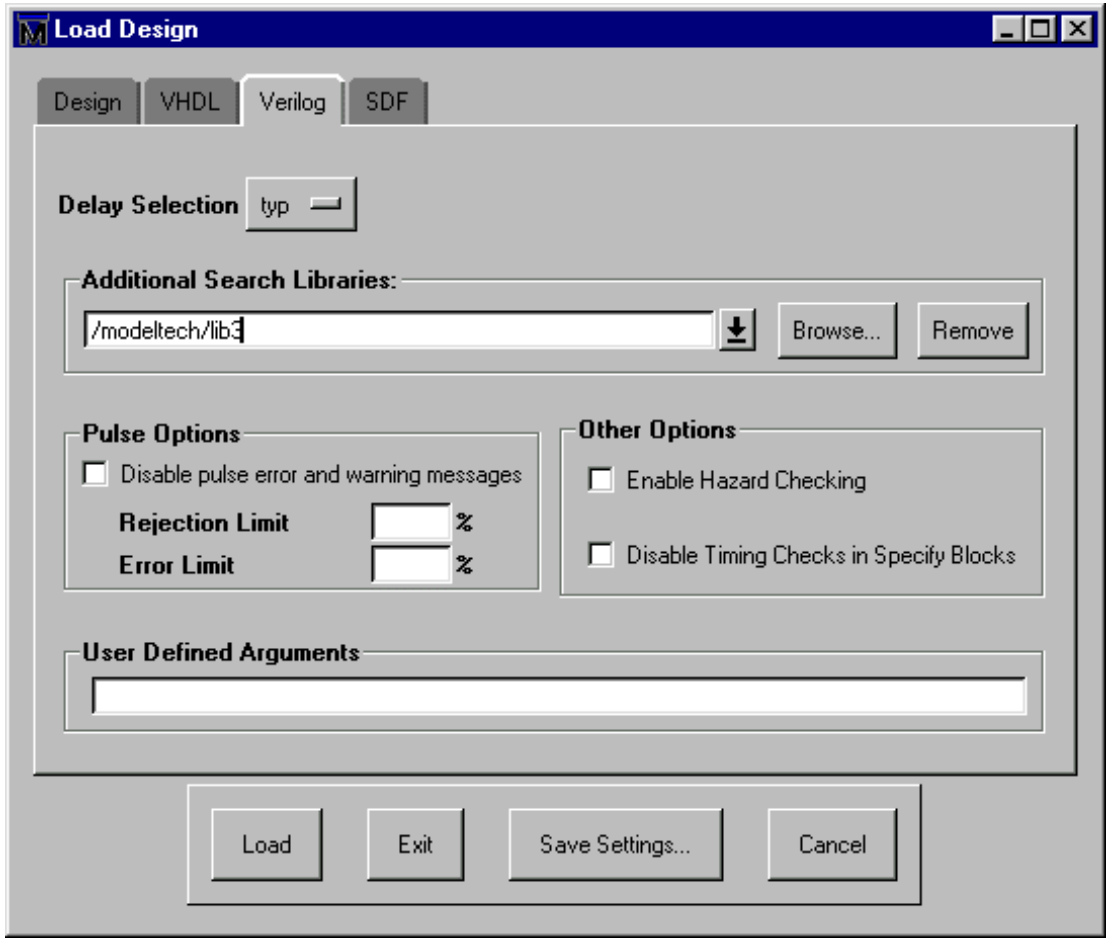
- **Disable Timing Checks** (+notimingchecks)
Disables timing checks generated by VITAL models.
- **Use Vital 2.2b SDF Mapping** (-vital2.2b)
Selects SDF mapping for VITAL 2.2b (default is Vital95).
- **Disable Glitch Generation** (-noglitch)
Disables VITAL glitch generation.

**TEXTIO files**

- **STD_INPUT** (-std_input <filename>)
Specifies the file to use for the VHDL textio STD_INPUT file. Use the **Browse** button to locate a file within your directories.

- **STD_OUTPUT** (-std_output <filename>)
  Specifies the file to use for the VHDL textio STD_OUTPUT file. Use the **Browse** button to locate a file within your directories.

## Verilog settings page



The **Verilog** page includes these options:

- **Delay Selection** (+mindelays | +typdelays | +maxdelays)
  Use the drop-down menu to select timing for min:typ:max expressions.
  Also see: "Timing check disabling" (4-46).
- **Additional Search Libraries** (-L <library_name>)
  Specifies one or more libraries to search for the design unit(s) you wish to simulate. Type in a library name or use the **Browse** button to locate a library within your directories. All specified libraries are added to the drop-down list; remove the currently selected library from the list with the **Remove** button. Make certain your selection is a valid Model*Sim* library - it must include an *_info* file and must have been created from Model*Sim*'s **vlib** command (CR-140).

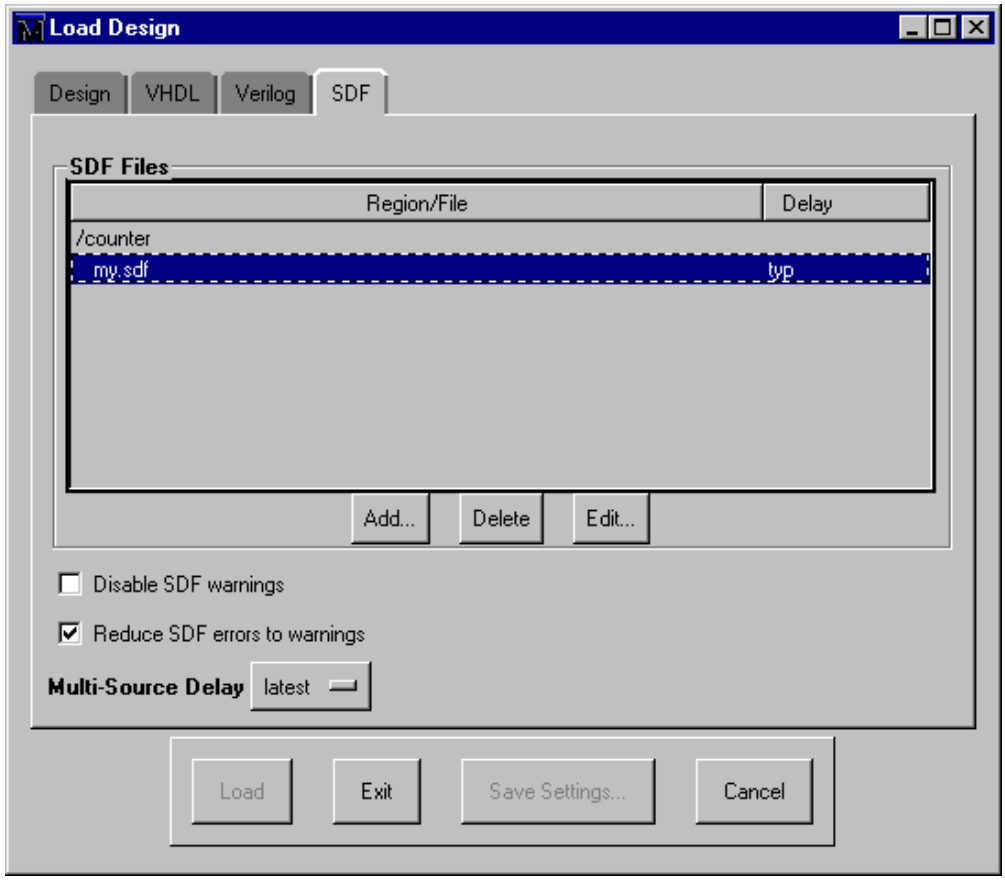
**Pulse Options**

- **Disable pulse error and warning messages** (+no_pulse_msg)
  Disables path pulse error warning messages.
- **Rejection Limit** (+pulse_r/<percent>)
  Sets module path pulse rejection limit as percentage of path delay.
- **Error Limit** (+pulse_e/<percent>)
  Sets module path pulse error limit as percentage of path delay.

**Other Options**

- **Enable Hazard Checking** (-hazards)
  Enables hazard checking in Verilog modules.
- **Disable Timing Checks in Specify Blocks** (+notimingchecks)
  Disables the timing check system tasks ($setup, $hold,...) in specify blocks.
- **User Defined Arguments** (+<plusarg>)
  Arguments are preceded with "+", making them accessible by the Verilog PLI routine **mc_scan_plusargs**. The values specified in this field must have a "+" preceding them or VSIM may incorrectly parse them.

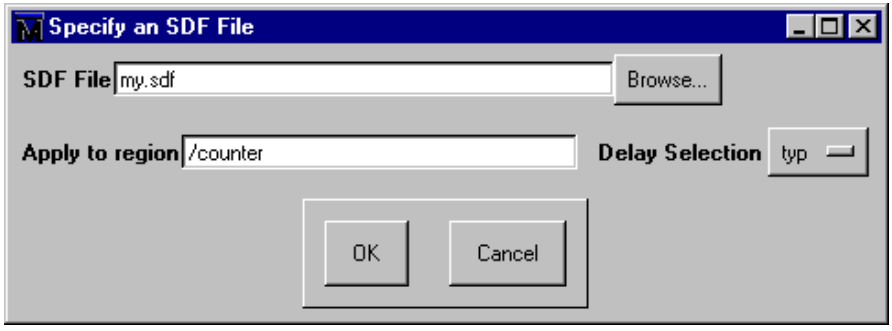
## SDF settings page




The **SDF** (Standard Delay Format) page includes these options:

**SDF Files**

The **Add** button opens a dialog box that allows you to specify the SDF files to load for the current simulation; files are then added to the **Region/File** list. You may also select a file on the listing to **Delete** or **Edit** (opens the dialog box below).




From the **Specify an SDF File** dialog box you can set the following options.

- **SDF file** ([<region>] = <sdf_filename>)
  Specifies the SDF file to use for annotation. Use the **Browse** button to locate a file within your directories.
- **Apply to region**
  Specifies the design region to use with the selected SDF options.
- **Delay Selection** (-sdfmin | -sdftyp | -sdfmax)
  Drop-down menu selects delay timing (min, typ or max) to be used from the specified SDF file. See also, "Specifying SDF files for simulation" (8-218).

  The **OK** button places the specified SDF file and delay on the **Region/File** list; **Cancel** dismisses the dialog box without changes.

and

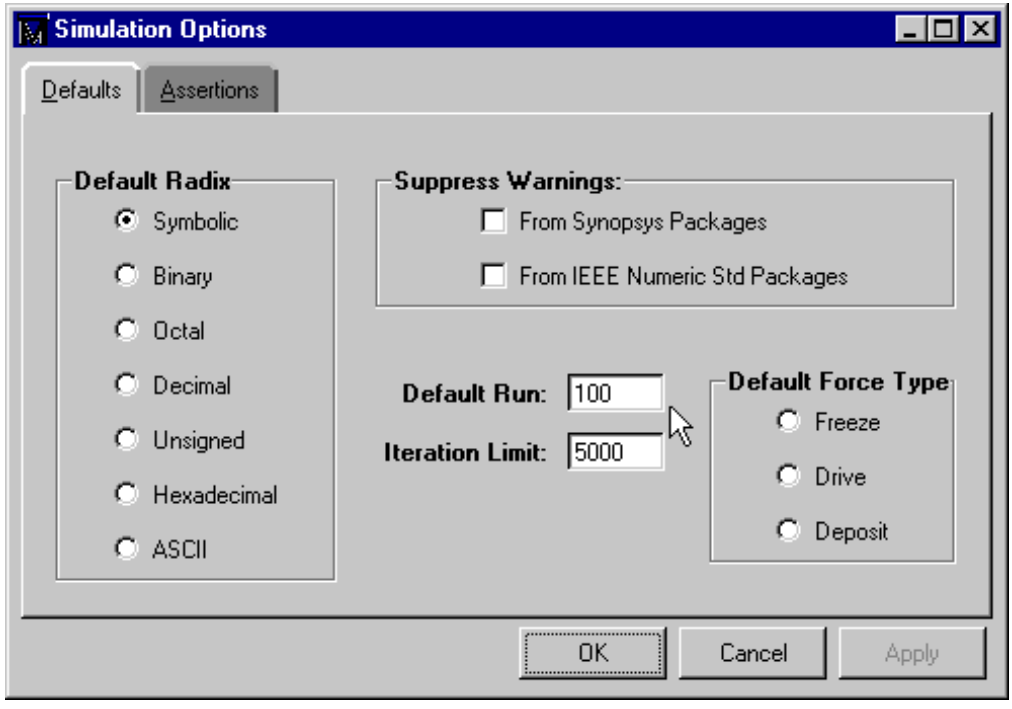
- **Disable warnings from SDF reader** (-sdfnowarn)
  Select to disable warnings from the SDF reader.
- **Reduce SDF errors to warnings** (-sdfnoerror)
  Change SDF errors to warnings so the simulation can continue.
- **Multi-Source Delay** (-multisource_delay <sdf_option>)
  Drop-down menu allows selection of **max**, **min** or **latest** delay. Controls how multiple PORT or INTERCONNECT constructs that terminate at the same port are handled. By default, the Module Input Port Delay (MIPD) is set to the **max** value encountered in the SDF file. Alternatively, you may choose the **min** or **latest** of the values.

## Setting default simulation options

Use the **Options > Simulation...** menu selection to bring up the **Simulation Options** dialog box shown below. Options you may set for the current simulation include: default radix, default force type, default run length, iteration limit, warning suppression, and break on assertion specifications. **OK** accepts the changes made and closes the dialog box. **Apply** makes the changes with the dialog box open so you can test your settings. **Cancel** closes the dialog box and makes no changes. The options found on each page are detailed below.

**Note:** Changes made in the **Simulation Options** dialog box are the default for the current simulation only. Options can be saved as the default for future simulations by editing the simulator control variables in the *modelsim.ini* file; the variables to edit are noted in the text below. You can use the **notepad** (CR-70) to edit the variables in *modelsim.ini* if you wish. See also, "Projects and system initialization" (3-35) for more information.

### Default settings page

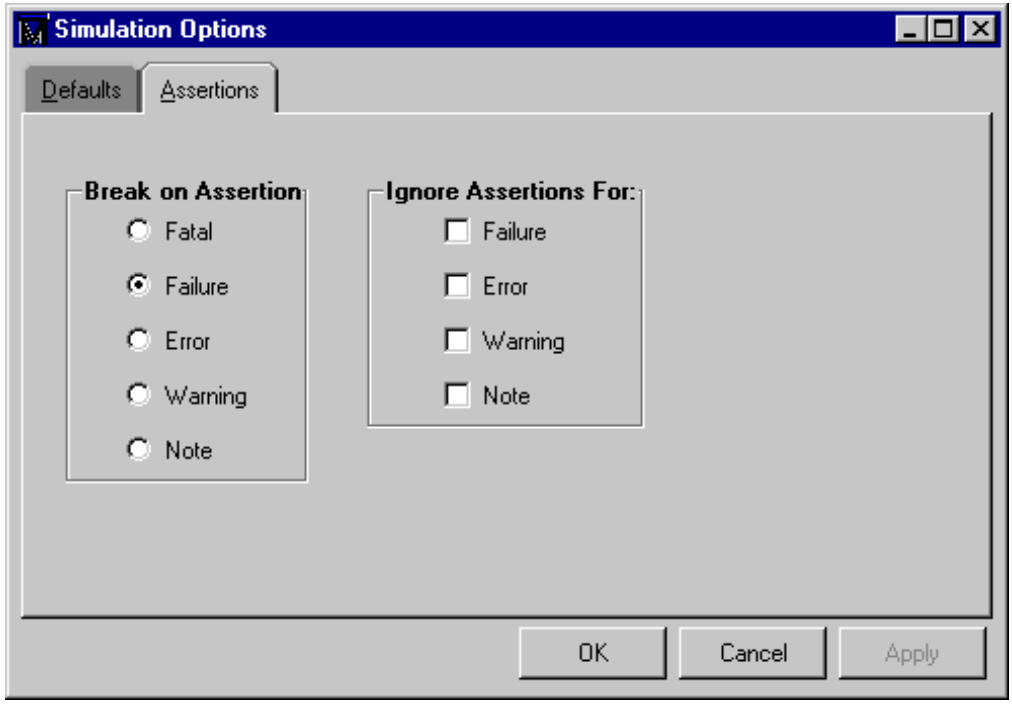


The **Default** page includes these options:

- **Default Radix**
  Sets the default radix for the current simulation run. You can also use the **radix** (CR-81) command to set the same temporary default. A permanent default can be set by editing the DefaultRadix (A-263) in the *modelsim.ini* file. The chosen radix is used for all commands (**force** (CR-56), **examine** (CR-51), **change** (CR-34) are examples) and for displayed values in the Signals, Variables, Dataflow, List, and Wave windows.
- **Default Force Type**
  Selects the default force type for the current simulation. Edit the DefaultForceKind (A-263) in the *modelsim.ini* to set a permanent default.
- **Suppress Warnings**
  Selecting **From IEEE Numeric Std Packages** suppresses warnings generated within the accelerated numeric_std and numeric_bit packages. Edit the NumericStdNoWarnings (A-264) in the *modelsim.ini* to set a permanent default.

  Selecting **From Synopsys Packages** suppresses warnings generated within the accelerated Synopsys std_arith packages. The permanent default can be set in the *modelsim.ini* file with the StdArithNoWarnings (A-265).
- **Default Run**
  Sets the default run length for the current simulation. A permanent default can be set in the *modelsim.ini* file with the RunLength (A-265).
- **Iteration Limit**
  Sets a limit on the number of deltas within the same simulation time unit to prevent infinite looping. A permanent iteration limit default can be set in the *modelsim.ini* file with the IterationLimit (A-264).

Assertion settings page




The **Assertions** page includes these options:

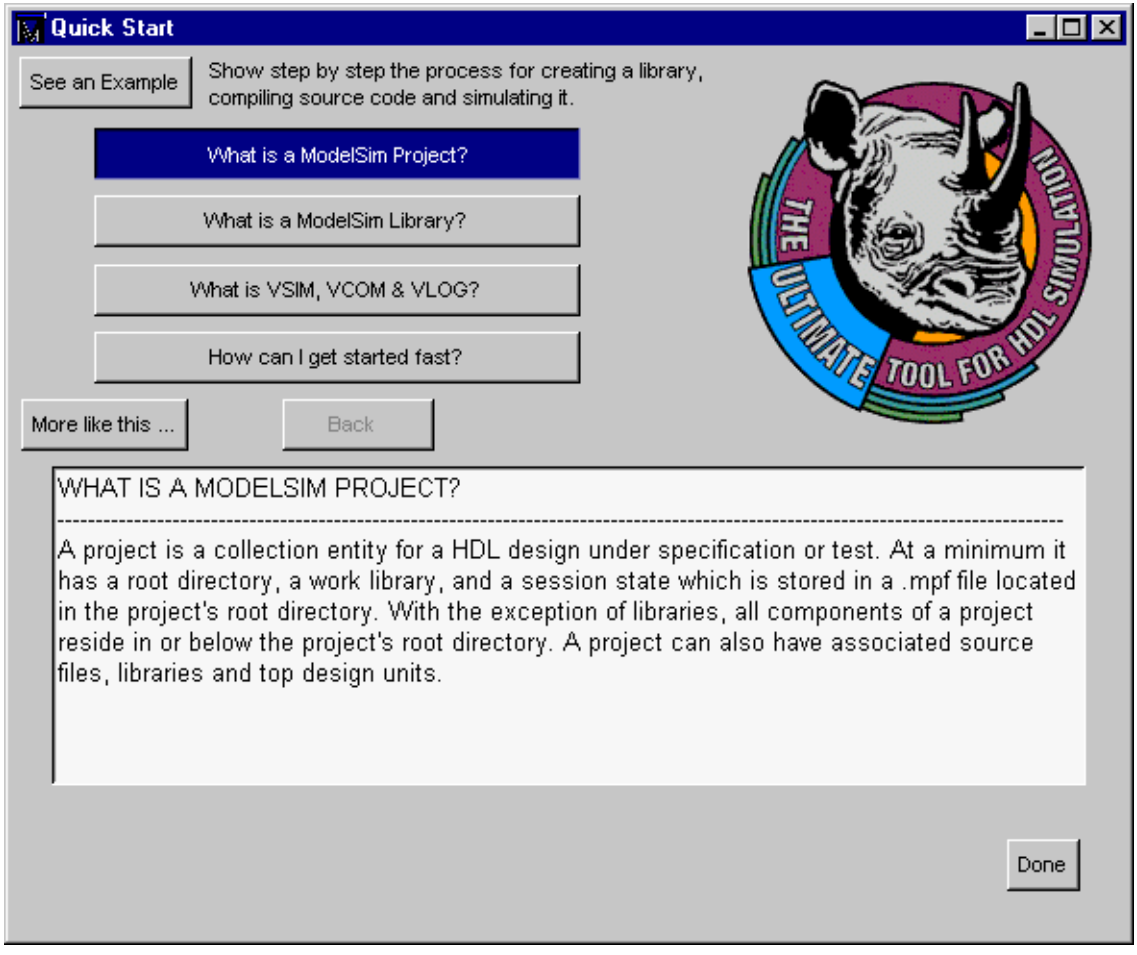
- **Break on Assertion**
  Selects the assertion severity that will stop simulation. Edit the BreakOnAssertion (A-263) in the *modelsim.ini* to set a permanent default.
- **Ignore Assertions For**
  Selects the assertion type to ignore for the current simulation. Multiple selections are possible. Edit the IgnoreFailure, IgnoreError, IgnoreWarning, or IgnoreNote (A-264) variables in the *modelsim.ini* to set a permanent default.

  When an assertion type is ignored, no message will be printed, nor will the simulation halt (even if break on assertion is set for that type).

**Note:** Assertions that appear within an instantiation or configuration port map clause or resolution function will not stop the simulation regardless of the severity level of the assertion.

## Model*Sim* Quick Start

The Model*Sim* Quick Start guide is available by clicking the **Quick Start** button in the Welcome to Model*Sim* window or by selecting **Help > Quick Start Menu** from the Main window. The Quick Start guide includes three examples (select the See an Example button) that shows the process for creating a project, creating a library, compiling source code and simulating the code.


Quick Start
See an Example
Show step by step the process for creating a library, compiling source code and simulating it.
What is a ModelSim Project?
What is a ModelSim Library?
What is VSIM, VCOM & VLOG?
How can I get started fast?
More like this ...
Back
THE ULTIMATE TOOL FOR HDL SIMULATION
WHAT IS A MODELSIM PROJECT?
A project is a collection entity for a HDL design under specification or test. At a minimum it has a root directory, a work library, and a session state which is stored in a .mpf file located in the project's root directory. With the exception of libraries, all components of a project reside in or below the project's root directory. A project can also have associated source files, libraries and top design units.
Done

Use the Quick Start guide to find online answers to the following questions:

- What is a Model*Sim* Project?
- What is a Model*Sim* Library?
- What is VSIM, VCOM & VLOG?
- How can I get started fast?
- How do I create a project in Model*Sim*?
- How do I specify design components of a project?
- How do I compile my project?
- How do I simulate my project?
- How do I stop working on a project?
- How do I modify a project?
- When do project setting changes take effect?
- What does enabling auto update of project settings mean?
- How do I use a project's settings with the command line tools?

# 8 - Standard Delay Format (SDF) Timing Annotation

## Chapter contents


SDF and ModelSim XE . . . . . 218
Specifying SDF files for simulation . . . . . 218
- Instance specification . . . . . 218
- SDF specification with the GUI . . . . . 219
- Errors and warnings . . . . . 219

VHDL VITAL SDF . . . . . 220
- SDF to VHDL generic matching . . . . . 220
- Resolving errors . . . . . 221

Verilog SDF . . . . . 222
- The $sdf_annotate system task . . . . . 222
- SDF to Verilog construct matching . . . . . 223
- Optional edge specifications . . . . . 227
- Optional conditions . . . . . 228
- Rounded timing values . . . . . 229

SDF for Mixed VHDL and Verilog Designs . . . . . 229

Interconnect delays . . . . . 229

Troubleshooting . . . . . 230

Obtaining the SDF specification . . . . . 232


This chapter discusses Model*Sim*'s implementation of SDF (Standard Delay Format) timing annotation. Included are sections on VITAL SDF and Verilog SDF, plus troubleshooting.

Verilog and VHDL VITAL timing data may be annotated from SDF files by using the simulator's built-in SDF annotator.

**Note:** Please see "SDF and ModelSim XE" (8-218) for Model*Sim* XE-specific information about SDF.

### SDF and ModelSim XE

For Model*Sim* XE, SDF timing annotation can only be applied to the Xilinx libraries shown below; all other libraries will simulate without annotation. The following mappings are defined in XE's *modelsim.ini* file:

```
simprim = $MODEL_TECH/../xilinx/vhdl/simprim
logiblox = $MODEL_TECH/../xilinx/vhdl/logiblox
unisim = $MODEL_TECH/../xilinx/vhdl/unisim
unisim5K = $MODEL_TECH/../xilinx/vhdl/unisim5K
simprims_ver = $MODEL_TECH/../xilinx/verilog/simprims
uni3000 = $MODEL_TECH/../xilinx/verilog/uni3000
unisims_ver = $MODEL_TECH/../xilinx/verilog/unisims_ver
uni5200 = $MODEL_TECH/../xilinx/verilog/uni5200
uni9000 = $MODEL_TECH/../xilinx/verilog/uni9000
```

## Specifying SDF files for simulation

Model*Sim* supports SDF versions 1.0 through 3.0. The simulator's built-in SDF annotator automatically adjusts to the version of the file. Use the following **vsim** (CR-148) command-line options to specify the SDF files, the desired timing values, and their associated design instances:

```
-sdfmin [<instance>=]<filename>
-sdftyp [<instance>=]<filename>
-sdfmax [<instance>=]<filename>
```

Any number of SDF files can be applied to any instance in the design by specifying one of the above options for each file. Use **-sdfmin** to select minimum, **-sdftyp** to select typical, and **-sdfmax** to select maximum timing values from the SDF file.

### Instance specification

The instance paths in the SDF file are relative to the instance that the SDF is applied to. Usually, this instance is an ASIC or FPGA model instantiated under a testbench. For example, to annotate maximum timing values from the SDF file *myasic.sdf* to an instance *u1* under a top-level named *testbench*, invoke the simulator as follows:

```
vsim -sdfmax /testbench/u1=myasic.sdf testbench
```

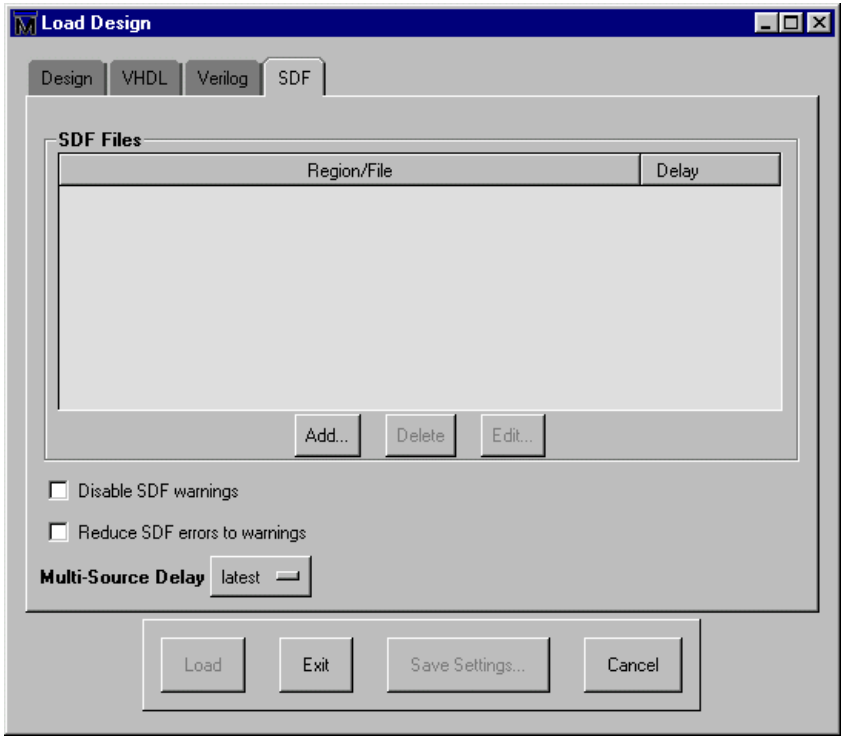
If the instance name is omitted then the SDF file is applied to the top-level. *This is usually incorrect* because in most cases the model is instantiated under a testbench or within a larger system level simulation. In fact, the design may have

several models, each having its own SDF file. In this case, specify an SDF file for each instance. For example,

```
vsim -sdfmax /system/u1=asic1.sdf -sdfmax /system/u2=asic2.sdf system
```

## SDF specification with the GUI

As an alternative to the command-line options, you may specify SDF files in the **Load Design** dialog box under the SDF tab.




This dialog box is presented if you invoke the simulator without any arguments or if you select "Load New Design..." under the simulator's file menu. For Verilog designs, you may also specify SDF files by using the **$sdf_annotate** system task. See "The $sdf_annotate system task" (8-222) for more details.

## Errors and warnings

Errors issued by the SDF annotator while loading the design prevent the simulation from continuing, whereas warnings do not. Use the **-sdfnoerror** option with **vsim** (CR-148) to change SDF errors to warnings so that the simulation can continue. Warning messages can be suppressed by using **vsim** with either the **-sdfnowarn** or **+nosdfwarn** options.

Another option is to use the **SDF** page from the **Load Design** dialog box (shown above). Select **Disable SDF warnings** (-sdfnowarn, or +nosdfwarn) to disable

warnings, or select **Reduce SDF errors to warnings** (-sdfnoerror) to change errors to warnings.

See "Troubleshooting" (8-230) for more information on errors and warnings, and how to avoid them.

## VHDL VITAL SDF

VHDL SDF annotation works on VITAL cells only. The IEEE 1076.4 VITAL ASIC Modeling Specification describes how cells must be written to support SDF annotation. Once again, the designer does not need to know the details of this specification because the library provider has already written the VITAL cells and tools that create compatible SDF files. However, the following summary is provided to help understand simulator error messages in case of user error or in case the vendor's SDF does not match the VITAL cells. For additional VITAL specification information see "Obtaining the VITAL specification and source code" (4-51).

### SDF to VHDL generic matching

An SDF file contains delay and timing constraint data for cell instances in the design. The annotator must locate the cell instances and the placeholders (VHDL generics) for the timing data. Each type of SDF timing construct is mapped to the name of a generic as specified by the VITAL modeling specification. The annotator locates the generic and updates it with the timing value from the SDF file. It is an error if the annotator fails to find the cell instance or the named generic. The following are examples of SDF constructs and their associated generic names:

| SDF construct | Matching VHDL generic name |
|---|---|
| (IOPATH a y (3)) | tpd_a_y |
| (IOPATH (posedge clk) q (1) (2)) | tpd_clk_q_posedge |
| (INTERCONNECT u1/y u2/a (5)) | tipd_a |
| (SETUP d (posedge clk) (5)) | tsetup_d_clk_noedge_posedge |
| (HOLD (negedge d) (posedge clk) (5)) | thold_d_clk_negedge_posedge |
| (SETUPHOLD d clk (5) (5)) | tsetup_d_clk & thold_d_clk |

| SDF construct | Matching VHDL generic name |
|---|---|
| (WIDTH (COND (reset==1'b0) clk) (5)) | tpw_clk_reset_eq_0 |

## Resolving errors

If the simulator finds the cell instance but not the generic then an error message is issued. For example,

```
ERROR: myasic.sdf(18):
Instance '/testbench/dut/u1' does not have a generic named
'tpd_a_y'
```

In this case, make sure that the design is using the appropriate VITAL library cells. If it is, then there is probably a mismatch between the SDF and the VITAL cells. You need to find the cell instance and compare its generic names to those expected by the annotator. Look in the VHDL source files directly or use the simulator's user interface to locate the information:

- Open the Structure window (7-158) and navigate to the instance named in the error message (you could try the Edit > Find menu option). Alternatively, use the **environment** command (CR-50) to select the instance. For example:

  ```
  env /testbench/dut/u1
  ```

- Open the Process window (7-142) and select the "In Region" mode (the default mode is "Active").
- Select a process in the Process window (usually the process named "vitalbehavior").
- Open the Variables window (7-161) to see all of the generics and their current values.

If none of the generic names look like VITAL timing generic names, then perhaps the VITAL library cells are not being used. If the generic names do look like VITAL timing generic names but don't match the names expected by the annotator, then there are several possibilities:

- The vendor's tools are not conforming to the VITAL specification.
- The SDF file was accidentally applied to the wrong instance. In this case, the simulator also issues other error messages indicating that cell instances in the SDF could not be located in the design.
- The vendor's library and SDF were developed for the older VITAL 2.2b specification. This version uses different name mapping rules. In this case, invoke **vsim** (CR-148) with the **-vital2.2b** option:

```
vsim -vital2.2b -sdfmax /testbench/u1=myasic.sdf testbench
```

For more information on resolving errors see "Troubleshooting" (8-230).

# Verilog SDF

Verilog designs may be annotated using either the simulator command-line options or the **$sdf_annotate** system task (also commonly used in other Verilog simulators). The command-line options annotate the design immediately after it is loaded, but before any simulation events take place. The **$sdf_annotate** task annotates the design at the time that it is called in the Verilog source code. This provides more flexibility than the command-line options.

## The $sdf_annotate system task

The syntax for **$sdf_annotate** is:

### Syntax

```
$sdf_annotate
      (["<sdffile>"], [<instance>], ["<config_file>"], ["<log_file>"],
      ["<mtm_spec>"], ["<scale_factor>"], ["<scale_type>"]);
```

### Arguments

`"<sdffile>"`
String that specifies the SDF file. Required.

`<instance>`
Hierarchical name of the instance to be annotated. Optional. Defaults to the instance where the $sdf_annotate call is made.

`"<config_file>"`
String that specifies the configuration file. Optional. Currently not supported, this argument is ignored.

`"<log_file>"`
String that specifies the logfile. Optional. Currently not supported, this argument is ignored.

`"<mtm_spec>"`
String that specifies delay selection. Optional. The allowed strings are "minimum", "typical", "maximum", and "tool_control". Case is ignored and the default is

"tool_control". The "tool_control" argument means to use the delay specified on the command line by +mindelays, +typdelays, or +maxdelays (defaults to +typdelays).

"<scale_factor>"
String that specifies delay scaling factors. Optional. The format is "<min_mult>:<typ_mult>:<max_mult>". Each multiplier is a real number that is used to scale the corresponding delay in the SDF file.

"<scale_type>"
String that overrides the <mtm_spec> delay selection. Optional. The <mtm_spec> delay selection is always used to select the delay scaling factor, but if a <scale_type> is specified, then it will determine the min/typ/max selection from the SDF file. The allowed strings are "from_min", "from_minimum", "from_typ", "from_typical", "from_max", "from_maximum", and "from_mtm". Case is ignored, and the default is "from_mtm", which means to use the <mtm_spec> value.

### Examples

Optional arguments may be omitted by using commas or by leaving them out if they are at the end of the argument list. For example, to specify only the SDF file and the instance it applies to:

```
$sdf_annotate("myasic.sdf", testbench.u1);
```

To also specify maximum delay values:

```
$sdf_annotate("myasic.sdf", testbench.u1, , , "maximum");
```

## SDF to Verilog construct matching

The annotator matches SDF constructs to corresponding Verilog constructs in the cells. Usually, the cells contain path delays and timing checks within specify blocks. For each SDF construct, the annotator locates the cell instance and updates each specify path delay or timing check that matches. An SDF construct may have multiple matches, in which case each matching specify statement is updated with the SDF timing value. SDF constructs are matched to Verilog constructs as follows:

**IOPATH** is matched to specify path delays or primitives:

| SDF | Verilog |
|---|---|
| (IOPATH (posedge clk) q (3) (4)) | (posedge clk => q) = 0; |
| (IOPATH a y (3) (4)) | buf u1 (y, a); |

The IOPATH construct usually annotates path delays. If the module contains no path delays, then all primitives that drive the specified output port are annotated.

**INTERCONNECT** and **PORT** are matched to input port:

| SDF | Verilog |
|---|---|
| (INTERCONNECT u1.y u2.a (5)) | input a; |
| (PORT u2.a (5)) | inout a; |

Both of these constructs identify a module input or inout port and create an internal net that is a delayed version of the port. This is called a Module Input Port Delay (MIPD). All primitives, specify path delays, and specify timing checks connected to the original port are reconnected to the new MIPD net.

**PATHPULSE** and **GLOBALPATHPULSE** are matched to specify path delays:

| SDF | Verilog |
|---|---|
| (PATHPULSE a y (5) (10)) | (a => y) = 0; |
| (GLOABLPATHPULSE a y (30) (60)) | (a => y) = 0; |

If the input and output ports are omitted in the SDF, then all path delays are matched in the cell.

**DEVICE** is matched to primitives or specify path delays:

| SDF | Verilog |
|---|---|
| (DEVICE y (5)) | and u1(y, a, b); |
| (DEVICE y (5)) | (a => y) = 0; (b => y) = 0; |

If the SDF cell instance is a primitive instance, then that primitive's delay is annotated. If it is a module instance, then all specify path delays are annotated that drive the output port specified in the DEVICE construct (all path delays are annotated if the output port is omitted). If the module contains no path delays, then all primitives that drive the specified output port are annotated (or all primitives that drive any output port if the output port is omitted).

**SETUP** is matched to $setup and $setuphold:

| SDF | Verilog |
|---|---|
| (SETUP d (posedge clk) (5)) | $setup(d, posedge clk, 0); |
| (SETUP d (posedge clk) (5)) | $setuphold(posedge clk, d, 0, 0); |

**HOLD** is matched to $hold and $setuphold:

| SDF | Verilog |
|---|---|
| (HOLD d (posedge clk) (5)) | $hold(posedge clk, d, 0); |
| (HOLD d (posedge clk) (5)) | $setuphold(posedge clk, d, 0, 0); |

**SETUPHOLD** is matched to $setup, $hold, and $setuphold:

| SDF | Verilog |
|---|---|
| (SETUPHOLD d (posedge clk) (5) (5)) | $setup(d, posedge clk, 0); |
| (SETUPHOLD d (posedge clk) (5) (5)) | $hold(posedge clk, d, 0); |
| (SETUPHOLD d (posedge clk) (5) (5)) | $setuphold(posedge clk, d, 0, 0); |

**RECOVERY** is matched to $recovery:

| SDF | Verilog |
|---|---|
| (RECOVERY (negedge reset) (posedge clk) (5)) | $recovery(negedge reset, posedge clk, 0); |

**REMOVAL** is matched to $removal:

| SDF | Verilog |
|---|---|
| (REMOVAL (negedge reset) (posedge clk) (5)) | $removal(negedge reset, posedge clk, 0); |

**RECREM** is matched to $recovery, $removal, and $recrem:

| SDF | Verilog |
|---|---|
| (RECREM (negedge reset) (posedge clk) (5) (5)) | $recovery(negedge reset, posedge clk, 0); |
| (RECREM (negedge reset) (posedge clk) (5) (5)) | $removal(negedge reset, posedge clk, 0); |
| (RECREM (negedge reset) (posedge clk) (5) (5)) | $recrem(negedge reset, posedge clk, 0); |

**SKEW** is matched to $skew:

| SDF | Verilog |
|---|---|
| (SKEW (posedge clk1) (posedge clk2) (5)) | $skew(posedge clk1, posedge clk2, 0); |

**WIDTH** is matched to $width:

| SDF | Verilog |
|---|---|
| (WIDTH (posedge clk) (5)) | $width(posedge clk, 0); |

**PERIOD** is matched to $period:

| SDF | Verilog |
|---|---|
| (PERIOD (posedge clk) (5)) | $period(posedge clk, 0); |

**NOCHANGE** is matched to $nochange:

| SDF | Verilog |
|---|---|
| (NOCHANGE (negedge write) addr (5) (5)) | $nochange(negedge write, addr, 0, 0); |

## Optional edge specifications

Timing check ports and path delay input ports may have optional edge specifications. The annotator uses the following rules to match edges:

- A match occurs if the SDF port does not have an edge.
- A match occurs if the specify port does not have an edge.
- A match occurs if the SDF port edge is identical to the specify port edge.
- A match occurs if explicit edge transitions in the specify port edge overlap with the SDF port edge.

These rules allow SDF annotation to take place even if there is a difference between the number of edge-specific constructs in the SDF file and the Verilog specify block. For example, the Verilog specify block may contain separate setup timing checks for a falling and rising edge on data with respect to clock, while the SDF file may contain only a single setup check for both edges:

| SDF | Verilog |
|---|---|
| (SETUP data (posedge clock) (5)) | $setup(posedge data, posedge clk, 0); |
| (SETUP data (posedge clock) (5)) | $setup(negedge data, posedge clk, 0); |

In this case, the cell accommodates more accurate data than can be supplied by the tool that created the SDF file, and both timing checks correctly receive the same value. Likewise, the SDF file may contain more accurate data than the model can accommodate

| SDF | Verilog |
|---|---|
| (SETUP (posedge data) (posedge clock) (4)) | $setup(data, posedge clk, 0); |
| (SETUP (negedge data) (posedge clock) (6)) | $setup(data, posedge clk, 0); |

In this case, both SDF constructs are matched and the timing check receives the value from the last one encountered.

Timing check edge specifiers may also use explicit edge transitions instead of posedge and negedge. However, the SDF file is limited to posedge and negedge. The explicit edge specifiers are 01, 0x, 10, 1x, x0, and x1. The set of [01, 0x, x1] is equivalent to posedge, while the set of [10, 1x, x0] is equivalent to negedge. A match occurs if any of the explicit edges in the specify port match any of the explicit edges implied by the SDF port. For example,

| SDF | Verilog |
|---|---|
| (SETUP data (posedge clock) (5)) | $setup(data, edge[01, 0x] clk, 0); |

## Optional conditions

Timing check ports and path delays may have optional conditions. The annotator uses the following rules to match conditions:

- A match occurs if the SDF does not have a condition.
- A match occurs for a timing check if the SDF port condition is semantically equivalent to the specify port condition.
- A match occurs for a path delay if the SDF condition is lexically identical to the specify condition.

Timing check conditions are limited to very simple conditions, therefore the annotator can match the expressions based on semantics. For example,

| SDF | Verilog |
|---|---|
| (SETUP data (COND (reset!=1) (posedge clock)) (5)) | $setup(data, posedge clk &&& (reset==0), 0); |

The conditions are semantically equivalent and a match occurs. In contrast, path delay conditions may be complicated and semantically equivalent conditions may not match. For example,

| SDF | Verilog |
|---|---|
| (COND (r1 \|\| r2) (IOPATH clk q (5))) | if (r1 \|\| r2) (clk => q) = 5; // matches |
| (COND (r1 \|\| r2) (IOPATH clk q (5))) | if (r2 \|\| r1) (clk => q) = 5; // does not match |

The annotator does not match the second condition above because the order of r1 and r2 are reversed.

### Rounded timing values

The SDF **TIMESCALE** construct specifies time units of values in the SDF file. The annotator rounds timing values from the SDF file to the time precision of the module that is annotated. For example, if the SDF TIMESCALE is 1ns and a value of .016 is annotated to a path delay in a module having a time precision of 10ps (from the timescale directive), then the path delay receives a value of 20ps. The SDF value of 16ps is rounded to 20ps. Interconnect delays are rounded to the time precision of the module that contains the annotated MIPD.

## SDF for Mixed VHDL and Verilog Designs

Annotation of a mixed VHDL and Verilog design is very flexible. VHDL VITAL cells and Verilog cells may be annotated from the same SDF file. This flexibility is available only by using the simulator's SDF command-line options. The Verilog $sdf_annotate system task can annotate Verilog cells only. See the **vsim** command (CR-148) for more information on SDF command-line options.

## Interconnect delays

An interconnect delay represents the delay from the output of one device to the input of another. This type of delay is modeled in the receiving device as a delay from an input port to an internal node. In VHDL VITAL this node is explicitly declared, whereas in Verilog it is automatically created by the simulator and is not visible to the user interface.

Timing checks are performed on the interconnect delayed versions of input ports. This may result in misleading timing constraint violations because the ports may satisfy the constraint while the delayed versions may not. If the simulator seems to report incorrect violations, be sure to account for the effect of interconnect delays.

Since an interconnect delay is modeled as a single delay between an input port and an internal node, there is no convenient way to handle interconnect delays from multiple outputs to a single input. For both VHDL VITAL and Verilog the default is to use the value of the maximum encountered delay in the SDF file. Optionally, you may choose the minimum or latest value of the multiple delays with the **vsim** (CR-148) **-multisource_delay** option:

```
-multisource_delay min|max|latest
```

# Troubleshooting

Several common mistakes in SDF annotation are outlined below.

## Specifying the wrong instance

By far, the most common mistake in SDF annotation is to specify the wrong instance to the simulator's SDF options. The most common case is to leave off the instance altogether, which is the same as selecting the top-level design unit. This is almost always wrong because the instance paths in the SDF are relative to the ASIC or FPGA model, which is usually instantiated under a top-level testbench. A common example for both VHDL and Verilog test benches is provided below. For simplicity, the test benches do nothing more than instantiate a model that has no ports.

### VHDL testbench

```
entity testbench is end;

architecture only of testbench is
    component myasic
    end component;
begin
    dut : myasic;
end;
```

### Verilog testbench

```
module testbench;
    myasic dut();
endmodule
```

The name of the model is *myasic* and the instance label is *dut*. For either testbench, an appropriate simulator invocation might be:

```
vsim -sdfmax /testbench/dut=myasic.sdf testbench
```

Optionally, you may leave off the name of the top-level:

```
vsim -sdfmax /dut=myasic.sdf testbench
```

The important point is to select the instance that the SDF is intended for. If the model is deep within the design hierarchy, an easy way to find the instance name is to first invoke the simulator without SDF options, open the structure window, navigate to the model instance, select it, and enter the **environment** command (CR-50). This command displays the instance name that should be used in the SDF command-line option.

### Mistaking a component or module name for an instance label

Another common error is to specify the component or module name rather than the instance label. For example, the following invocation is wrong for the above testbenches:

```
vsim -sdfmax /testbench/myasic=myasic.sdf testbench
```

This results in the following error message:

```
ERROR: myasic.sdf:
The design does not have an instance named '/testbench/myasic'.
```

### Forgetting to specify the instance

If you leave off the instance altogether, then the simulator issues a message for each instance path in the SDF that is not found in the design. For example,

```
vsim -sdfmax myasic.sdf testbench
```

Results in:

```
ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u1'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u2'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u3'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u4'

ERROR: myasic.sdf:
Failed to find INSTANCE '/testbench/u5'

WARNING: myasic.sdf:
This file is probably applied to the wrong instance.

WARNING: myasic.sdf:
Ignoring subsequent missing instances from this file.
```

After annotation is done, the simulator issues a summary of how many instances were not found and possibly a suggestion for a qualifying instance:

```
WARNING: myasic.sdf:
Failed to find any of the 358 instances from this file.

WARNING: myasic.sdf:
Try instance '/testbench/dut' - it contains all instance paths from
this file.
```

The simulator recommends an instance only if the file was applied to the top-level and a qualifying instance is found one level down.

Also see "Resolving errors" (8-221) for specific VHDL VITAL SDF troubleshooting.

## Obtaining the SDF specification

SDF specification is available from Open Verilog International:

Lynn Horobin
phone: (408)358-9510
fax: (408)358-3910
email: info@ovi.org
home page: http://www.ovi.org

# 9 - Value Change Dump (VCD) Files

## Chapter contents

ModelSim VCD commands and VCD tasks . . . . . 234

A VCD file from source to output . . . . . 234
VHDL source code . . . . . 234
VCD simulator commands . . . . . 235
VCD output . . . . . 235

This chapter explains Model Technology's Verilog VCD implementation for Model*Sim*.

The VCD file format is specified in the IEEE 1364 standard. It is an ASCII file containing header information, variable definitions, and variable value changes. VCD is in common use for Verilog designs, and is controlled by VCD system task calls in the Verilog source code. Model*Sim* provides simulator command equivalents for these system tasks and extends VCD support to VHDL designs; the Model*Sim* commands can be used on either VHDL or Verilog designs.

**Note:** If you need vendor-specific ASIC design-flow documentation that incorporates VCD, please contact your ASIC vendor.

## Model*Sim* VCD commands and VCD tasks

Model*Sim* VCD commands map to IEEE 1364 VCD system tasks and appear in the VCD file along with the results of those commands. The table below shows the mapping of the extended VCD commands to the IEEE 1364 keywords.

| VCD commands | VCD system tasks |
|---|---|
| **vcd add** (CR-98) | $dumpvars |
| **vcd checkpoint** (CR-99) | $dumpall |
| **vcd file** (CR-101) | $dumpfile |
| **vcd flush** (CR-102) | $dumpflush |
| **vcd limit** (CR-103) | $dumplimit |
| **vcd off** (CR-104) | $dumpoff |
| **vcd on** (CR-105) | $dumpon |

In addition to the commands above, the **vcd comment** command (CR-100) can be used to add comments to the VCD file.

## A VCD file from source to output

The following example shows the VHDL source, a set of simulator commands, and the resulting VCD output.

### VHDL source code

The design is a simple shifter device represented by the following VHDL source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SHIFTER_MOD is
    port (CLK, RESET, data_in   : IN STD_LOGIC;
        Q : INOUT STD_LOGIC_VECTOR(8 downto 0));

END SHIFTER_MOD ;
```

```
architecture RTL of SHIFTER_MOD is

begin
    process (CLK,RESET)
    begin
        if (RESET = '1') then
            Q <= (others => '0') ;
        elsif (CLK'event and CLK = '1') then
            Q <= Q(Q'left - 1 downto 0) & data_in ;
        end if ;
    end process ;
end ;
```

## VCD simulator commands

At simulator time zero, the designer executes the following commands and quits the simulator at time 1200:

```
vcd file output.vcd
vcd add -r *
force reset 1 0
force data_in 0 0

force clk 0 0
run 100
force clk 1 0, 0 50 -repeat 100
run 100
vcd off
force reset 0 0
force data_in 1 0
run 100
vcd on
run 850
force reset 1 0
run 50
vcd checkpoint
```

## VCD output

The VCD file created as a result of the preceding scenario would be called *output.vcd*. The following pages show how it would look.

VCD output

```
$comment
    File created using the following
command:
        vcd file output.vcd
$date
    Fri Apr 12 09:07:17 1996
$end
$version
    ModelSim EE/PLUS 5.1
$end
$timescale
    1ns
$end
$scope module shifter_mod $end
$var wire 1 ! clk $end
$var wire 1 " reset $end
$var wire 1 # data_in $end
$var wire 1 $ q [8] $end
$var wire 1 % q [7] $end
$var wire 1 & q [6] $end
$var wire 1 ’ q [5] $end
$var wire 1 ( q [4] $end
$var wire 1 ) q [3] $end
$var wire 1 * q [2] $end
$var wire 1 + q [1] $end
$var wire 1 , q [0] $end
$upscope $end
$enddefinitions $end
#0
$dumpvars
0!
1"
0#
0$
0%
0&
0’
0(
0)
0*
0+
0,
$end
#100
1!
#150
0!
#200
1!
$dumpo
ff
x!
x"
x#
x$
x%
x&
x’
x(
x)
x*
x+
x,
$end
#300
$dumpo
n
1!
0"
1#
0$
0%
```

```
0&
0’
0(
0)
0*
0+
1,
$end
#350
0!
#400
1!
1+
#450
0!
#500
1!
1*
#550
0!
#600
1!
1)
#650
0!
#700
1!
1(
#750
0!
#800
1!
1’
#850
0!
#900
1!
1&
#950
0!
#1000
1!
1%
#1050
0!
#1100
1!
1$
#1150
0!
1"
0$
0%
0&
0’
0(
0)
0*
0+
0,
#1200
1!
$dumpa
ll
1!
1"
1#
0$
0%
0&
0’
0(
0)
0*
0+
0,
$end
```

# 10 - Tcl and Model*Sim*

## Chapter contents


Tcl features within ModelSim . . . . . 240

Tcl References . . . . . 240

Tcl commands . . . . . 241
- Tcl command syntax . . . . . 242
- if command syntax . . . . . 245
- set command syntax . . . . . 246
- Command substitution . . . . . 247
- Command separator. . . . . 247
- Multiple-line commands . . . . . 247
- Evaluation order . . . . . 247
- Tcl relational expression evaluation . . . . . 248
- Variable substitution . . . . . 248
- System commands . . . . . 249

List processing . . . . . 249

VSIM Tcl commands . . . . . 250

ModelSim Tcl time commands . . . . . 250


This chapter provides an overview of Tcl (tool command language) as used with Model*Sim*. Additional Tcl and Tk (Tcl's toolkit) can be found through several Tcl online references (10-240).

Tcl is a scripting language for controlling and extending Model*Sim*. Within Model*Sim* you can develop implementations from Tcl scripts without the use of C code. Because Tcl is interpreted, development is rapid; you can generate and execute Tcl scripts on the fly without stopping to recompile or restart VSIM. In addition, if VSIM does not provide the command you need, you can use Tcl to create your own commands.

## Tcl features within Model*Sim*

Using Tcl with Model*Sim* gives you these features:

- command history (like that in C shells)
- full expression evaluation and support for all C-language operators
- a full range of math and trig functions
- support of lists and arrays
- regular expression pattern matching
- procedures
- the ability to define your own commands
- command substitution (that is, commands may be nested)

## Tcl References

### Tcl print references

Two sources of information about Tcl are *Tcl and the Tk Toolkit* by John K. Ousterhout, published by Addison-Wesley Publishing Company, Inc., and *Practical Programming in Tcl and Tk by* Brent Welch published by Prentice Hall.

### Tcl online references

The following are a few of the many Tcl references available:

- When using Model*Sim,* select **Help > Tcl Man Pages** from the Main window menus.
- Tcl man pages are also available at: www.elf.org/tcltk-man-html/contents.htm
- Tcl/Tk general information is available from the Tcl/Tk Consortium: www.tclconsortium.org
- The Scriptics Corporation, John Ousterhout's company (the original Tcl developer): www.scriptics.com.

## Tcl commands

The Tcl commands are listed below. For complete information on Tcl commands use the Main window menu selection: **Help > Tcl Man Pages**, or refer to one of the Tcl/Tk resources noted above. Also see "Preference variables located in TCL files" (A-271) for information on Tcl variables.

| | | | | |
|---|---|---|---|---|
| append | array | break | case | catch |
| cd | close | concat | continue | eof |
| error | eval | exec | expr | file |
| flush | for | foreach | format | gets |
| glob | global | history | if | incr |
| info | insert | join | lappend | list |
| llength | lindex | lrange | lreplace | lsearch |
| lsort | open | pid | proc | puts |
| pwd | read | regexp | regsub | rename |
| return | scan | seek | set | split |
| string | switch | tell | time | trace |
| source | unset | uplevel | upvar | while |

**Note:** Model*Sim* command names that conflict with Tcl commands have been renamed or have been replaced by Tcl commands. See the list below:

| Previous Model*Sim* command | Command changed to (or replaced by) |
|---|---|
| continue | **run** (CR-86) with the **-continue** option |
| format list \| wave | **write format** (CR-167) with either list or wave specified |

| Previous Model*Sim* command | Command changed to (or replaced by) |
|---|---|
| if | replaced by the Tcl **if** command, see "if command syntax" (10-245) for more information |
| list | **add list** (CR-19) |
| nolist \| nowave | **delete** (CR-40) with either list or wave specified |
| set | replaced by the Tcl **set** command, see "set command syntax" (10-246) for more information |
| source | **vsource** (CR-159) |
| wave | **add wave** (CR-23) |

## Tcl command syntax

The former Model*Sim* commands, **if** and **set** are now Tcl commands. You should understand Tcl command syntax before using these commands. The syntax, especially for the **if** command, may be unfamiliar.

The following rules define the syntax and semantics of the Tcl language. Both the if command syntax (10-245) and set command syntax (10-246) follow the general discussion of Tcl command syntax.

1. A Tcl script is a string containing one or more commands. Semi-colons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

2. A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

3. Words of a command are separated by white space (except for newlines, which are command separators).

4. If the first character of a word is double-quote (""") then the word is terminated by the next double-quote character. If semi-colons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary

characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.

5 If the first character of a word is an open brace ("{") then the word is terminated by the matching close brace ("}"). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

6 If a word contains an open bracket ("[") then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket ("]"). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

7 If a word contains a dollar-sign ("$") then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

`$name`

Name is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

`$name(index)`

Name gives the name of an array variable and index gives the name of an element within that array. Name must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of index.

`${name}`

Name is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

**8** If a backslash ("\") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

**\a**

Audible alert (bell) (0x7).

**\b**

Backspace (0x8).

**\f**

Form feed (0xc).

**\n**

Newline (0xa).

**\r**

Carriage-return (0xd).

**\t**

Tab (0x9).

**\v**

Vertical tab (0xb).

**\<newline>**whiteSpace

A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

****

Backslash ("\").

*****ooo*

The digits ooo (one, two, or three of them) give the octal value of the character.

**\x***hh*

The hexadecimal digits hh give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

**9** If a hash character ("#") appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

**10** Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

**11** Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

## if command syntax

The Tcl **if** command executes scripts conditionally. Note that in the syntax below the "?" indicates an optional argument.

### Syntax

**if** *expr1* ?**then**? *body1* **elseif** *expr2* ?**then**? *body2* **elseif** ... ?**else**? ?*bodyN*?

### Description

The **if** command evaluates *expr1* as an expression. The value of the expression must be a boolean (a numeric value, where 0 is false and anything is true, or a string value such as **true** or **yes** for true and **false** or **no** for false); if it is true then body1 is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The **then** and **else** arguments are optional "noise words" to make the command easier to read. There may be any number of **elseif** clauses, including zero. *BodyN* may also be omitted as long as **else** is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

## set command syntax

The Tcl **set** reads and writes variables. Note that in the syntax below the "?" indicates an optional argument.

### Syntax

**set** *varName ?value?*

### Description

Returns the value of variable *varName*. If value is specified, then set the value of *varName* to value, creating a new variable if one doesn't already exist, and return its value. If *varName* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *varName* refers to a scalar variable. Normally, *varName* is unqualified (does not include the names of any containing namespaces), and the variable of that name in the current namespace is read or written. If *varName* includes namespace qualifiers (in the array name if it refers to an array element), the variable in the specified namespace is read or written.

If no procedure is active, then *varName* refers to a namespace variable (global variable if the current namespace is the global namespace). If a procedure is active, then *varName* refers to a parameter or local variable of the procedure unless the global command was invoked to declare *varName* to be global, or unless a Tcl **variable** command was invoked to declare varName to be a namespace variable.

### More Tcl commands

All Tcl commands are documented from within Model*Sim*. Select **Help** > **Tcl Help** from the **Main** window menus.

## Command substitution

Placing a command in square brackets [ ] will cause that command to be evaluated first and its results returned in place of the command. An example is:

```
set a 25
set b 11
set c 3
echo "the result is [expr ($a + $b)/$c]"
```

will output:

```
"the result is 12"
```

This feature allows VHDL variables and signals, and Verilog nets and registers to be accessed using:

```
[examine -<radix> name]
```

The %name substitution is no longer supported. Everywhere %name could be used, you now can use [examine -value -<radix> name] which allows the flexibility of specifying command options. The radix specification is optional.

## Command separator

A semicolon character (;) works as a separator for multiple commands on the same line. It is not required at the end of a line in a command sequence.

## Multiple-line commands

With Tcl, multiple-line commands can be used within macros and on the command line. The command line prompt will change (as in a C shell) until the multiple-line command is complete.

In the example below, note the way the opening brace { is at the end of the if and else lines. This is important because otherwise the Tcl scanner won't know that there is more coming in the command and will try to execute what it has up to that point, which won't be what you intend.

```
if { [exa sig_a] == "0011ZZ"}  {
    echo "Signal value matches"
    do macro_1.do
} else {
    echo "Signal value fails"
    do macro_2.do  }
```

## Evaluation order

An important thing to remember when using Tcl is that anything put in curly brackets {} is not evaluated immediately. This is important for if-then-else, procedures, loops, and so forth.

## Tcl relational expression evaluation

When you are comparing values, the following hints may be useful:

- Tcl stores all values as strings, and will convert certain strings to numeric values when appropriate. If you want a literal to be treated as a numeric value, don't quote it.

  ```
  if {[exa var_1] == 345}...
  ```

  The following will also work:

  ```
  if {[exa var_1] == "345"}...
  ```

- However, if a literal cannot be represented as a number, you *must* quote it, or Tcl will give you an error. For instance:

  ```
  if {[exa var_2] == 001Z}...
  ```

  will give an error.

  ```
  if {[exa var_2] == "001Z"}...
  ```

  will work okay.

- Don't quote single characters in single quotes:

  ```
  if {[exa var_3] == 'X'}...
  ```

  will give an error

  ```
  if {[exa var_3] == "X"}...
  ```

  will work okay.

- For the equal operator, you must use the C operator "==" . For not-equal, you must use the C operator "!=".

## Variable substitution

When a $<var_name> is encountered, the Tcl parser will look for variables that have been defined either by VSIM or by you, and substitute the value of the variable.

**Note:** Tcl is case sensitive for variable names.

To access environment variables, use the construct:

```
$env(<var_name>)
echo My user name is $env(USER)
```

Environment variables can also be set using the env array:

```
set env(SHELL) /bin/csh
```

See "Simulator state variables" (A-283) for more information about VSIM-defined variables.

### System commands

To pass commands to the DOS window, use the Tcl exec command:

```
echo The date is [exec date]
```

## List processing

In Tcl a "list" is a set of strings in curly braces separated by spaces. Several Tcl commands are available for creating lists, indexing into lists, appending to lists, getting the length of lists and shifting lists. These commands are:

| Command syntax | Description |
|---|---|
| **lappend** var_name val1 val2 ... | appends val1, val2, etc. to list var_name |
| **lindex** list_name index | return the index-th element of list_name; the first element is 0 |
| **linsert** list_name index val1 val2 ... | inserts val1, val2, etc. just before the index-th element of list_name |
| **list** val1, val2 ... | returns a Tcl list consisting of val1, val2, etc. |
| **llength** list_name | returns the number of elements in list_name |
| **lrange** list_name first last | returns a sublist of list_name, from index first to index last; first or last may be "end", which refers to the last element in the list |
| **lreplace** list_name first last val1, val2, ... | replaces elements first through last with val1, val2, etc. |

Two other commands, **lsearch** and **lsort,** are also available for list manipulation. See the Tcl man pages (Main window: **Help > Tcl Man Pages**) for more information on these commands.

See also the Model*Sim* Tcl command: **lecho** (CR-61)

## VSIM Tcl commands

These additional VSIM commands enhance the interface between Tcl and Model*Sim*, Only brief descriptions are provided here; for more information and command syntax see the "ModelSim Commands" (CR-11).

| Command | Description |
|---|---|
| **alias** (CR-28) | creates a new Tcl procedure that evaluates the specified commands; used to create a user-defined alias |
| **lecho** (CR-61) | takes one or more Tcl lists as arguments and pretty-prints them to the VSIM Main window |
| **lshift** (CR-64) | takes a Tcl list as argument and shifts it in-place one place to the left, eliminating the 0th element |
| **lsublist** (CR-65) | returns a sublist of the specified Tcl list that matches the specified Tcl glob pattern |
| **project** (CR-76) | echoes to the VSIM Main window the current names and values of all environment variables |

## Model*Sim* Tcl time commands

Model*Sim* Tcl time commands make simulator-time-based values available for use within other Tcl procedures.

Time values may optionally contain a units specifier where the intervening space is also optional. If the space is present, the value must be quoted (e.g. 10ns, "10 ns"). Time values without units are taken to be in the UserTimeScale. Return values are always in the current Time Scale Units. All time values are converted to a 64-bit integer value in the current Time Scale. This means that values smaller than the current Time Scale will be truncated to 0.

## Conversions

| Command | Description |
|---|---|
| intToTime <intHi32> <intLo32> | converts two 32-bit pieces (high and low order) into a 64-bit quantity (Time in Model*Sim* is a 64-bit integer) |
| RealToTime <real> | converts a <real> number to a 64-bit integer in the current Time Scale |
| scaleTime <time> <scaleFactor> | returns the value of <time> multiplied by the <scaleFactor> integer |

## Relations

| Command | Description |
|---|---|
| eqTime <time> <time> | evaluates for equal |
| neqTime <time> <time> | evaluates for not equal |
| gtTime <time> <time> | evaluates for greater than |
| gteTime <time> <time> | evaluates for greater than or equal |
| ltTime <time> <time> | evaluates for less than |
| lteTime <time> <time> | evaluates for less than or equal |

All relation operations return 1 or 0 for true or false respectively and are suitable return values for TCL conditional expressions. For example,

```
if {[eqTime $Now 1750ns]} {
        ...
    }
```

### Arithmetic

| Command | Description |
|---|---|
| addTime <time> <time> | add time |
| divTime <time> <time> | 64-bit integer divide |
| mulTime <time> <time> | 64-bit integer multiply |
| subTime <time> <time> | subtract time |

# A - ModelSim Variables

## Appendix contents


Variable settings report . . . . . 254

Personal preferences . . . . . 254

Returning to the original ModelSim defaults . . . . . 254

Environment variables . . . . . 255

Preference variables located in INI and MPF files . . . . . 259
- [Library] library path variables . . . . . 259
- [vcom] VHDL compiler control variables . . . . . 260
- [vlog] Verilog compiler control variables . . . . . 262
- [vsim] simulator control variables . . . . . 262
- [Project] project file section (MPF files only) . . . . . 266
- Setting variables in INI / MPF files . . . . . 267
- Variable functions . . . . . 267

Preference variables located in TCL files . . . . . 271
- Viewing the default preference file (pref.tcl) . . . . . 271
- Individual preference variables . . . . . 275
- Setting Tcl preference variables . . . . . 276

Simulator state variables . . . . . 283


This appendix documents the following types of Model*Sim* variables:

- **environment variables**
  Variables referenced and set according to operating system conventions. Environment variables prepare the Model*Sim* environment prior to simulation.
- **Model*Sim* preference variables**
  Variables used to control compiler or simulator functions (usually in .tcl files) and modify the appearance of the Model*Sim* GUI (usually in INI and MPF files).
- **simulator state variable**s
  Variables that provide feedback on the state of the current simulation.

## Variable settings report

The **report** command (CR-82) returns a list of current settings for either the simulator state, or simulator control variables. Use the following commands at either the Model*Sim* or VSIM prompt:

```
report simulator state
report simulator control
```

## Personal preferences

There are several preferences stored by Model*Sim* on a personal bases, independent of *modelsim.ini* or *modelsim.tcl* files. These preferences are stored in the Windows Registry under HKEY_CURRENT_USER\Software\Model Technology Incorporated\ModelSim.

- **cwd**
  History of the last five working directories (pwd). This history appears in the Main window File menu.
- **phst**
  Project History
- **pinit**
  Project Initialization state (one of: Welcome | OpenLast | NoWelcome)
- **printersetup**
  All setup parameters related to Printing (i.e., current printer, etc.)

The HKEY_CURRENT_USER key is unique for each user Login on Windows NT.

## Returning to the original Model*Sim* defaults

If you would like to return Model*Sim*'s interface to its original state, simply rename or delete the existing *modelsim.tcl* and *modelsim.ini* files. Model*Sim* will use *pref.tcl* for GUI preferences and make a copy of *<install_dir>/modeltech/modelsim.ini* to use the next time VSIM is invoked without an existing project (if you start a new project the new MPF file will use the settings in the new *modelsim.ini* file.

# Environment variables

Before compiling or simulating, several environment variables may be set to provide the functions described in the table below. The variables are in the *autoexec.bat* file on Windows 95/98 machines, and set through the System control panel on NT machines. The LM_LICENSE_FILE variable is required, all others are optional.

## ModelSim Environment Variables

| Variable | Description |
|---|---|
| DOPATH | used by VSIM to search for simulator command files (do files); consists of a colon-separated (semi-colon for Windows) list of paths to directories; optional; this variable can be overridden by the DOPATH (A-275) .tcl file variable |
| EDITOR | specifies the editor to invoke with the **edit** command (CR-48) |
| HOME | used by VSIM to look for an optional graphical preference file and optional location map file; see: "Preference variables located in INI and MPF files" (A-259) and "Modeling memory in VHDL" (C-293) |
| LM_LICENSE_FILE | used by the ModelSim license file manager to find the location of the license file; may be a colon-separated (semi-colon for Windows) set of paths, including paths to other vendor license files; REQUIRED |
| MODEL_TECH | set by all ModelSim tools to the directory in which the binary executables reside; **YOU SHOULD NOT SET THIS VARIABLE** |
| MODEL_TECH_TCL | used by VSIM to find Tcl libraries for: Tcl/Tk 8.0, Tix, and VSIM; defaults to <install_dir>/../tcl; may be set to an alternate path |
| MGC_LOCATION_MAP | used by ModelSim tools to find source files based on easily reallocated "soft" paths; optional; see: "Modeling memory in VHDL" (C-293); also see the Tcl variables: SourceDir (A-275), and SourceMap (A-275) |
| MTI_TF_LIMIT | limits the size of the VSOUT temp file (generated by the VSIM kernel); the value of the variable is the size of k-bytes; TMPDIR (below) controls the location of this file, STDOUT controls the name; default = 10, 0 = no limit |
| PLIOBJS | used by VSIM to search for PLI object files for loading; consists of a space-separated list of file or path names; optional |

| Variable | Description |
|---|---|
| STDOUT | the VSOUT temp file (generated by the simulator kernel) is deleted when the simulator exits; the file is not deleted if you specify a filename for VSOUT with STDOUT; specifying a name and location (use TMPDIR) for the VSOUT file will also help you locate and delete the file in event of a crash (an unnamed VSOUT file is not deleted after a crash either) |
| TMP | specifies the path to a tempnam() generated file (VSOUT) containing all stdout from the simulation kernel; optional |
| MODELSIM | used by all Model*Sim* tools to find the *modelsim.ini* file; consists of a path including the file name; optional. An alternative use of this variable is to set it to the path of a project file (<Project_Root_Dir>/ <Project_Name>.mpf). This allows you to use project settings with command line tools. However, if you do this, the .mpf file will replace modelsim.ini as the initialization file for all ModelSim tools. |
| MODELSIM_TCL | used by VSIM to look for an optional graphical preference file |

## Setting environment variables in Windows

In addition to the predefined variables shown above, you can define your own environment variables. This example shows a user-defined library path variable that can be referenced by the **vmap** command to add library mapping to the *modelsim.ini* file.

### Using Windows 95/98

Open and edit the *autoexec.bat* file by adding this line:

```
set MY_PATH=\temp\work
```

Restart Windows to initialize the new variable.

### Using Windows NT

Right-click the My Computer icon and select Properties, then select the Environment tab of the System Properties control panel. Add the new variable to these fields: Variable:MY_PATH and Value:\temp\work.

Click Set and Apply to initialize the variable (you don’t need to restart NT).

### Library mapping with environment variables

Once the **MY_PATH** variable is set, you can use it with the **vmap** command (CR-147) to add library mappings to the current *modelsim.ini* file.

If you're using the **vmap** command from DOS prompt type:

```
vmap MY_VITAL %MY_PATH%
```

If you're using **vmap** from Model*Sim*/VSIM prompt type:

```
vmap MY_VITAL \$MY_PATH
```

If you used DOS **vmap**, this line will be added to the *modelsim.ini*:

```
MY_VITAL = c:\temp\work
```

If **vmap** is used from Model*Sim*/VSIM prompt, the *modelsim.ini* will be modified with this line:

```
 MY_VITAL = $MY_PATH
```

You can easily add additional hierarchy to the path. For example,

```
vmap MORE_VITAL %MY_PATH%\more_path\and_more_path

vmap MORE_VITAL \$MY_PATH\more_path\and_more_path
```

**Note:** The "$" character in the examples above is Tcl syntax that precedes a variable. The "\" character is an escape character that keeps the variable from being evaluated during the execution of **vmap**.)

## Referencing environment variables within Model*Sim*

There are two ways to reference environment variables within Model*Sim*. Environment variables are allowed in a **FILE** variable being opened in VHDL. For example,

```
entity test is end;
use std.textio.all;

architecture only of test is

begin
    process
        FILE in_file : text is in "$ENV_VAR_NAME";
    begin
        wait;
    end process;
end;
```

Environment variables may also be referenced from the Model*Sim* command line or in macros using the Tcl **env** array mechanism:

```
echo "$env(ENV_VAR_NAME)"
```

## Removing temp files (VSOUT)

The *VSOUT* temp file is the communication mechanism between the simulator kernel and the Model*Sim* GUI. In normal circumstances the file is deleted when the simulator exits. If Model*Sim* crashes, however, the temp file must be deleted manually. Specifying the location of the temp file with **TMPDIR** (above) will help you locate and remove the file.

**Note:** There is one environment variable, **MODEL_TECH**, that you cannot — and should not — set. **MODEL_TECH** is a special variable set by Model Technology software. Its value is the name of the directory from which the **vcom** compiler or **vsim** simulator was invoked. **MODEL_TECH** is used by the other Model Technology tools to find the libraries.

# Preference variables located in INI and MPF files

Model*Sim* initialization (INI) and project (MPF) files contain control variables that specify reference library paths, and compiler and simulator settings. When first created, the MPF project file includes all of the variables from the current *modelsim.ini* file plus an additional [Project] section for project-specific variables.

For information on creating Model*Sim* projects see *Chapter 3 - Projects and system initialization*.

The following tables list the variables by section, and in order of their appearance within the INI/MPF file:


| INI and MPF file sections |
|---|
| [Library] library path variables (A-259) |
| [vcom] VHDL compiler control variables (A-260) |
| [vlog] Verilog compiler control variables (A-262) |
| [vsim] simulator control variables (A-262) |
| [Project] project file section (MPF files only) (A-266) |


## [Library] library path variables

| Variable name | Value range | Purpose |
|---|---|---|
| std | any valid path; may include environment variables | sets path to the VHDL STD library; default is <install_dir>/../std |
| ieee | any valid path; may include environment variables | sets path to the library containing IEEE and Synopsys arithmetic packages; default is <install_dir>/../ieee |
| verilog | any valid path; may include environment variables | sets path to the library containing VHDL/Verilog type mappings; default is <install_dir>/../verilog |
| std_developerskit | any valid path; may include environment variables | sets path to the libraries for MGC standard developer's kit |

## [Library] library path variables

| Variable name | Value range | Purpose |
|---|---|---|
| synopsys | any valid path; may include environment variables | sets path to the accelerated arithmetic packages |

## [vcom] VHDL compiler control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| VHDL93 | 0, 1 | if 1, turns on VHDL-1993 | off (0) |
| Show_source | 0, 1 | if 1, shows source line containing error | off (0) |
| Show_VitalChecksWarnings | 0, 1 | if 0, turns off VITAL compliance-check warnings | on (1) |
| Show_Warning1 | 0, 1 | if 0, turns off unbound-component warnings | on (1) |
| Show_Warning2 | 0, 1 | if 0, turns off process-without-a-wait-statement warnings; default is on | on (1) |
| Show_Warning3 | 0, 1 | if 0, turns off null-range warnings | on (1) |
| Show_Warning4 | 0, 1 | if 0, turns off no-space-in-time-literal warnings | on (1) |
| Show_Warning5 | 0, 1 | if 0, turns off multiple-drivers-on-unresolved-signal warnings | on (1) |
| Optimize_1164 | 0, 1 | if 0, turns off optimization for IEEE std_logic_1164 package | on (1) |

## [vcom] VHDL compiler control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| Explicit | 0, 1 | if 1, turns on resolving of ambiguous function overloading in favor of the "explicit" function declaration (not the one automatically created by the compiler for each type declaration) | off (0) |
| NoVitalCheck | 0, 1 | if 1, turns off VITAL compliance checking | off (0) |
| IgnoreVitalErrors | 0, 1 | if 1, ignores VITAL compliance checking errors | off (0) |
| NoDebug | 0, 1 | if 1, turns off inclusion of debugging info within design units | off (0) |
| NoVital | 0, 1 | if 1, turns off acceleration of the VITAL packages | off (0) |
| Hazard | 0, 1 | if 1, turns on Verilog hazard checking (order-dependent accessing of global vars) | off (0) |
| Quiet | 0, 1 | if 1, turns off "loading..." messages | off (0) |
| CheckSynthesis | 0, 1 | if 1, turns on limited synthesis rule compliance checking; checks only signals used (read) by a process | off (0) |
| ScalarOpts | 0, 1 | if 1, activate optimizations on expressions that don’t involve signals, waits or function/procedure/task invocations | off (0) |

## [vlog] Verilog compiler control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| Hazard | 0, 1 | if 1, turns on Verilog hazard checking (order-dependent accessing of global vars) | off (0) |
| NoDebug | 0, 1 | if 1, turns off inclusion of debugging info within design units | off (0) |
| Quiet | 0, 1 | if 1, turns off "loading..." messages | off (0) |
| ScalarOpts | 0, 1 | if 1, activate optimizations on expressions that don't involve signals, waits or function/ procedure/task invocations | off (0) |
| Show_source | 0, 1 | if 1, shows source line containing error | off (0) |
| UpCase | 0, 1 | if 1, turns on converting regular Verilog identifiers to uppercase. Allows case insensitivity for module names; see also "Verilog-XL compatible compiler options" (5-60) | off (0) |

## [vsim] simulator control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| AssertFile | any valid filename | alternative file for storing assertion messages | commented out (;) |

## [vsim] simulator control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| AssertionFormat | see purpose | sets the message to display after a break on assertion; message formats include:<br>%S - severity level<br>%R - report message<br>%T - time of assertion<br>%D - delta<br>%I - instance or region pathname (if available)<br>%% - print '%' character | "** %S: %R\n Time: %T Iteration: %D%I\n" |
| BreakOnAssertion | 0-4 | defines severity of assertion that causes a simulation break (0 = note, 1 = warning, 2 = error, 3 = failure, 4 = fatal) | 3 |
| CheckpointCompressMode | 0, 1 | if 1, checkpoint files are written in compressed format | on (1) |
| CommandHistory | any valid filename | set the name of a file to store the Main window command history | commented out (;) |
| ConcurrentFileLimit | any positive integer | controls the number of VHDL files open concurrently; this number should be less than the current limit setting for max file descriptors; 0 = unlimited | 40 |
| DatasetSeparator | any single character | the dataset separator for fully-rooted contexts, for example sim:/top; must not be the same character as PathSeparator | : |
| DefaultForceKind | freeze, drive, or deposit | defines the kind of force used when not otherwise specified | commented out (;) |
| DefaultRadix | symbolic, binary, octal, decimal, unsigned, hexadecimal, ascii | any radix may be specified as a number or name, i.e., binary can be specified as binary or 2 | symbolic |

## [vsim] simulator control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| DelayFileOpen | 0, 1 | if 1, open VHDL87 files on first read or write, else open files when elaborated | off (0) |
| GenerateFormat | %s__%d | control the format of a generate statement label (don't quote it) | commented out (;) |
| IgnoreError | 0,1 | if 1, ignore assertion errors | off (0) |
| IgnoreFailure | 0,1 | if 1, ignore assertion failures | off (0) |
| IgnoreNote | 0,1 | if 1, ignore assertion notes | off (0) |
| IgnoreWarning | 0,1 | if 1, ignore assertion warnings | off (0) |
| IterationLimit | positive integer | limit on simulation kernel iterations during one time delta | 5000 |
| License | any single <license_option> | if set, controls Model*Sim* license file search; license options include:<br>nomgc - excludes MGC licenses<br>nomti - excludes MTI licenses<br>vlog - only use VLOG license<br>vhdl - only use VHDL license<br>plus - only use PLUS license<br>noqueue - do not wait in license queue if no license see also the **vsim** command (CR-148) <license_option> | search all licenses |
| NumericStdNoWarnings | 0, 1 | if 1, warnings generated within the accelerated numeric_std and numeric_bit packages are suppressed | off (0) |
| PathSeparator | any single character | used for hierarchical path names | / |

## [vsim] simulator control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| Resolution | fs, ps, ns, us, ms, sec - also 10x and 100x | simulator resolution; default is ns; this value must be less than or equal to the UserTimeUnit specified below; NOTE - if your delays are truncated, set the resolution smaller | ns |
| RunLength | positive integer | default simulation length in units specified by the UserTimeUnit variable | 100 |
| Start up | = do <DO filename>; any valid macro (do) file | specifies the VSIM startup macro; default is commented out; see the **do** command (CR-43) | commented out (;) |
| StdArithNoWarnings | 0, 1 | if 1, warnings generated within the accelerated Synopsys std_arith packages are suppressed | off (0) |
| TranscriptFile | any valid filename | file for saving command transcript; environment variables many be included in the path name; default is "transcript"; | transcript |
| UnbufferedOutput | 0, 1 | controls VHDL files open for write; 0 = Buffered, 1 = Unbuffered | 0 |
| UserTimeUnit | fs, ps, ns, us, ms, sec, min, hr | specifies the default units to use for the "<timesteps> [<time_units>]" argument to the **run** command (CR-86); NOTE - the value of this variable must be set equal to, or larger than, the current simulator resolution specified by the Resolution variable shown above | ns |
| Veriuser | one or more valid shared object | list of dynamically loaded objects for Verilog PLI applications; see "Using the Verilog PLI" (5-80) | commented out (;) |

## [vsim] simulator control variables

| Variable name | Value range | Purpose | Default |
|---|---|---|---|
| WaveSignalNameWidth | 0, positive or negative integer | controls the number of visible hierarchical regions of a signal name shown in the Wave window (7-164); the default value of zero displays the full name, a setting of one or above displays the corresponding level(s) of hierarchy | 0 |

## [Project] project file section (MPF files only)

| Variable name | Value range | Purpose |
|---|---|---|
| Src_Files | a bracketed, space-separated list of source file paths in the form of: {<pathname>} {<path with spaces>} | list all source files that have not yet been compiled into a library; the bracketed list allows spaces within paths (as does Windows); see the example MPF file below |
| Cur_Top_DUs | *NONE* or list of one or more top-level design units (DUs) consisting of a space-seperated list in the form of: <library>.<design_unit> | lists current or last top-level design unit(s) loaded; signifies that there is loadable design; *NONE* is the initial state and indicates that no auto design load should occur; a subset of Top_DUs |
| Top_DUs | one or more top level design units consisting of a space-seperated list in the form of: <library>.<design_unit> | lists all top-level design units that could be loaded for simulation |
| Work_Libs | a bracketed, space-seperated list of library paths in the form of: {<pathname>} {<path with spaces>} | complete list of all libraries local to this project; libraries are listed in compile order; the bracketed list allows spaces within paths (as does Windows); see the example MPF file below |
| <Lib>_script | path to a do script containing compiler instructions for recompiling a library <Lib> | the specified script is invoked to build the corresponding library <Lib> |

### Spaces in path names

For the Src_Files and Work_Libs variables, each element in the list is enclosed within curly braces ({}). This allows spaces inside elements (since Windows allows spaces inside path names). For example a source file list might look like:

```
Src_Files = {$MODELSIM_PROJECT/counter.v} {$MODELSIM_PROJECT/tb counter.v}
```

Where the file *tb counter.v* contains a space character between the "b" and "c".

## Setting variables in INI / MPF files

Edit the initialization or project file directly with any text editor to change or add a variable. The syntax for variables in the file is:

```
<variable> = <value>
```

Comments within the file are preceded with a semicolon ( ; ).

**Note:** The **vmap** command (CR-147) automatically modifies library mapping in the current INI / MPF file.

## Variable functions

Several, though not all, of the *modelsim.ini* variables are further explained below.

### Environment variables

You can use environment variables in your initialization files. Use a dollar sign ($) before the environment variable name.

### Examples

```
[Library]
work = $HOME/work_lib
test_lib = ./$TESTNUM/work
...
[vsim]
IgnoreNote = $IGNORE_ASSERTS
IgnoreWarning = $IGNORE_ASSERTS
IgnoreError = 0
IgnoreFailure = 0
```

Tip:

There is one environment variable, MODEL_TECH, that you cannot — and should not — set. MODEL_TECH is a special variable set by Model Technology software. Its value is

the name of the directory from which the VCOM compiler or VSIM simulator was invoked. MODEL_TECH is used by the other Model Technology tools to find the libraries.

### Hierarchical library mapping

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the Model*Sim* tools don't find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the "others" clause.

### Examples

```
[Library]
asic_lib = /cae/asic_lib
work = my_work
others = /install_dir/modeltech/modelsim.ini
```

Tip:

Since the file referred to by the others clause may itself contain an others clause, you can use this feature to chain a set of hierarchical INI files.

### Creating a transcript file

A feature in the system initialization file allows you to keep a record of everything that occurs in the transcript: error messages, assertions, commands, command outputs, etc. To do this, set the value for the TranscriptFile line in the *modelsim.ini* file to the name of the file in which you would like to record the VSIM history. The size of this file can be controlled with the MTI_TF_LIMIT (A-255).

```
; Save the command window contents to this file
TranscriptFile = trnscrpt
```

### Using a startup file

The system initialization file allows you to specify a command or a *do* file that is to be executed after the design is loaded. For example:

```
; VSIM Startup command
Startup = do mystartup.do
```

The line shown above instructs VSIM to execute the commands in the macro file named *mystartup.do*.

```
; VSIM Startup command
Startup = run -all
```

The line shown above instructs VSIM to run until there are no events scheduled.

See the **do** command (CR-43) for additional information on creating do files.

### Turning off assertion messages

You can turn off assertion messages from your VHDL code by setting a switch in the *modelsim.ini* file. This option was added because some utility packages print a huge number of warnings.

```
[vsim]
IgnoreNote = 1
IgnoreWarning = 1
IgnoreError = 1
IgnoreFailure = 1
```

Messages may also be turned off with Tcl variables; see "Preference variables located in TCL files" (A-271).

### Turning off warnings from arithmetic packages

You can disable warnings from the synopsys and numeric standard packages by adding the following lines to the [vsim] section of the *modelsim.ini* file.

```
[vsim]
NumericStdNoWarnings = 1
StdArithNoWarnings = 1
```

Warnings may also be turned off with Tcl variables; see "Preference variables located in TCL files" (A-271).

### Force command defaults

The VSIM **force** command has **-freeze**, **-driver**, and **-deposit** options. When none of these is specified, then **-freeze** is assumed for unresolved signals and **-drive** is assumed for resolved signals. This is designed to provide compatibility with force files. But if you prefer **-freeze** as the default for both resolved and unresolved signals, you can change the defaults in the *modelsim.ini* file.

```
[vsim]
; Default Force Kind
; The choices are freeze, drive, or deposit
DefaultForceKind = freeze
```

### VHDL93

You can make the VHDL93 standard the default by including the following line in the *INI* file:

```
[vcom]
; Turn on VHDL1993 as the default (default is 0)
VHDL93 = 1
```

### Opening VHDL files

You can delay the opening of VHDL files with a entry in the *INI* file if you wish. Normally VHDL files are opened when the file declaration is elaborated. If the DelayFileOpen option is enabled, then the file is not opened until the first read or write to that file.

```
[vsim]
DelayFileOpen = 1
```

# Preference variables located in TCL files

Model*Sim* TCL preference variables give you control over fonts, colors, prompts, window positions and other simulator window characteristics. Preference files, which contain Tcl commands that set preference variables, are loaded before any windows are created, and so will affect all windows.

When Model*Sim* is invoked for the first time, default preferences are loaded from the *pref.tcl* file. Customized variable settings may be set from within the Model*Sim* GUI or by directly editing the preference file.

The default file for customized preferences is *modelsim.tcl*. If your preference file is not named *modelsim.tcl*, you must refer to it with the MODELSIM_TCL (A-256) environment variable.

## User-defined variables

Temporary user-defined variables can be created with the Tcl **set** command. Like simulator variables, user-defined variables are preceded by a dollar sign when referenced. To create a variable with the **set** command:

```
set user1 7
```

You can use the variable in a command like:

```
echo "user1 = $user1"
```

## Viewing the default preference file (pref.tcl)

This documentation covers the categories of preference variables found in the *pref.tcl* file. You can open the file in any text editor to see a complete listing of the default preferences. See "Setting Tcl preference variables" (A-276) before you change any of the preferences.

**Note:** If you do open and edit a TCL file, make sure you save it as plain text, otherwise it will not be properly interpreted by Model*Sim*.

## Preference variable arrays

Most preference variables are Tcl procedure lists (arrays), grouped by name within the TCL file. A unique array is defined for:

- all GUI defaults
- each VSIM window type
- the library browser
- the Code Coverage and Performance Analyzer windows
- logic value translations used in the List window (7-128) and Wave window (7-164)
- the **force** command (CR-56)

The most common variable array types are listed in the table below.

| Variable array type | Description |
|---|---|
| PrefDefault(<argument>) | handles all default GUI preferences such as font attributes, menu features, and colors |
| Pref<WindowName>(<argument>) | an array exists for each Model*Sim* window covering fonts, colors, and window-specific variables such as the (isTrigger) List window variable |
| Pref<WindowName>(geometry) | (geometry) variables are used for initial window positions with a new design; one variable for each window |
| PrefGeometry(<WindowName) | if you change the window positions and invoke this command:<br>`write pref ./modelsim.tcl`<br>this additional set of geometry preference variables is written to *modelsim.tcl* with a higher priority; on the next invocation of the **vsim** (CR-148) command you will get the newly-saved positions; (**write pref** saves all current preference setting) |
| PrefLibrary(<argument>) | library preferences allow you to add color to different design units within the library browser |

| Variable array type | Description |
|---|---|
| ListTranslateTable(<argument>) | ListTranslateTable specifies how various enumerations of various types map into the nine logic types that the List and Wave window know how to display. This mapping is used for vectors only; scalars are displayed with the original enum value. The following example values show that the std_logic_1164 types map in a one-to-one manner, and also shows mappings for boolean and Verilog types. You can add additional translations for your own user-defined types. |
| LogicStyleTable(<argument>) | LogicStyleTable variables allow you to control how each of the nine internal logic types are graphically displayed in the Wave window. For each of the nine internal logic types, a three-element Tcl list specifies: the line type, the line color, and the line vertical location; the line type may be Solid, OnOffDash, or DoubleDash. For vertical location, 0 is at the bottom of the waveform, 1 is at the middle, and 2 is at the top. |
| ForceTranslateTable(<argument>) | ForceTranslateTable is used only for vectors, and maps how a string of digits are mapped into enumerations. First, digits 0, 1, X and Z are mapped into LOGIC_0, LOGIC_1, LOGIC_X, and LOGIC_Z, respectively. Then the ForceTranslateTable is used to map from there to the enumeration appropriate for the type of signal being forced. |

## Main window preference variables

The Main window uses preference variables similar to other Model*Sim* window to conrol colors and fonts. The variables below control some additional functions.

| Variable | Description | Default |
|---|---|---|
| PrefMain(cmdHistory) | set the name of a file to store the Main window command history | history |
| PrefMain(DisplayDatasetPrefix) | turns dataset prefix viewing on or off, 1 displays the prefix and 0 turns the prefix off | 1, or dataset prefix is diplayed |
| PrefMain(file) | name of the file for saving transcript; an environment variable may be used | transcript |
| PrefMain(forceQuit) | determines whether you are prompted to confirm exiting Modelsim; when set to 0, you are prompted to confirm exit; when set to 1, you are not prompted to confirm exit (this variable cannot be set via the GUI) | 0 |
| PrefMain(linkWindows) | controls whether all Model*Sim* windows minimize with the Main window:<br>1 indicates windows are linked; 0 indicates windows are not linked (see note below this table). When linking is disabled, select Windows->Icon All to iconize all windows. | 1 for Windows NT/ 98<br><br>0 for UNIX |
| PrefMain(prompt1) | used as primary prompt | {VSIM [history nextid]>} |
| PrefMain(prompt2) | prompt used when no design is loaded | "ModelSim> " |
| PrefMain(prompt3) | prompt used when macro is interrupted | "VSIM(paused)> " |

## Individual preference variables

Though most preference variables occur in arrays, the following individual variables are also found in TCL files.

| Variable name | Value range | Purpose |
|---|---|---|
| DOPATH | a colon-separated list of paths to directories | used by VSIM to search for simulator command files (DO files); overrides the DOPATH (A-255) environment variable |
| PlotFilterResolution | 0.1 + | specifies the output resolution for the waveform postscript file; default is 0.2, which equals 600dpi resolution; 0.1 equals 1200dpi; 0.4 equals 300dpi, etc. |
| SourceDir | any valid path | a list of alternate directories to search for source files; separate multiple paths with a colon |
| SourceMap | any valid path | a Tcl associative array for mapping a particular source file path (index) to another source file path (value) |

## The addons variable

The PrefVsim(addOns) variable provides a user_hook that allows you to integrate add-ons with VSIM. Refer to any vendor-supplied instructions for specifics about connecting third-party add-ons to VSIM.

| Variable | Description | Example value |
|---|---|---|
| PrefVsim(addOns) | implicitly adds switches to VSIM invocation; useful for loading foreign libraries (see example value) | {-f "power_init $MGC_HOME/lib/libpwr.dl"} |

## Setting Tcl preference variables

Preference variable within TCL file may be set in one of two ways.

- Setting variables with the GUI (A-276)
- Directly editing preference files (A-281)

### Setting variables with the GUI

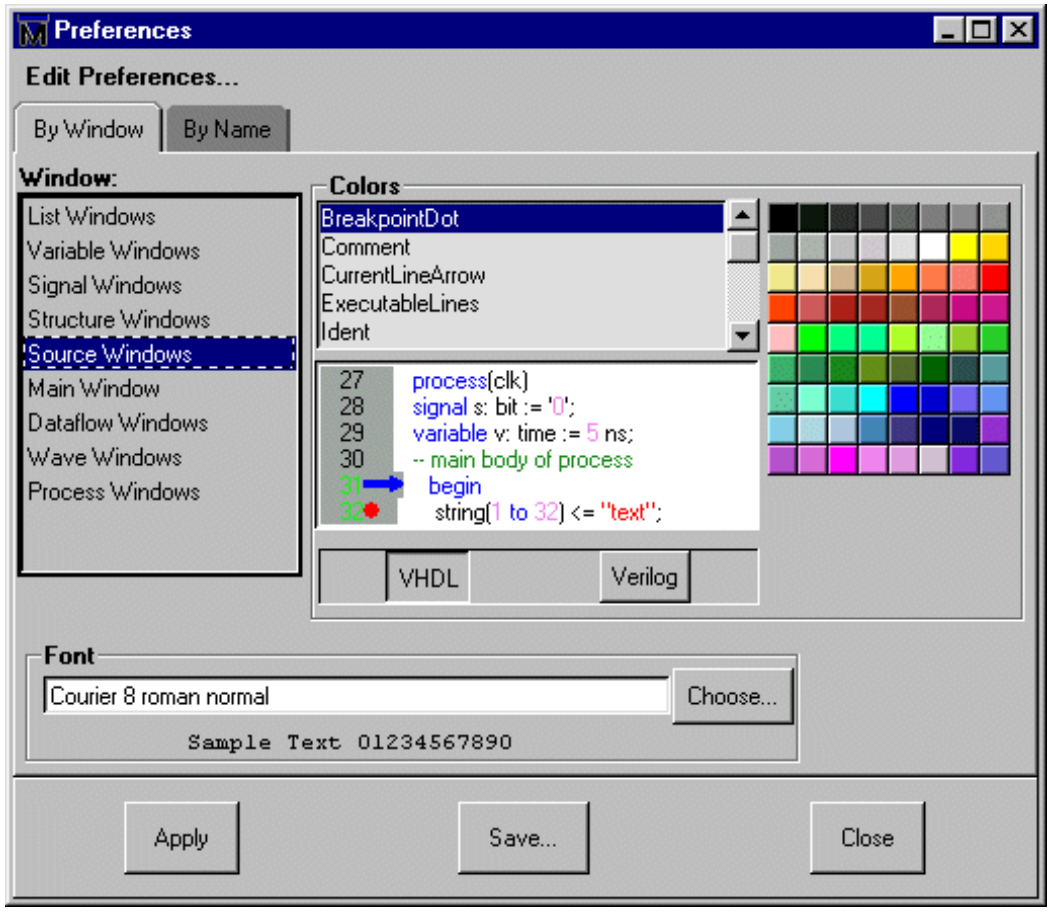
Use the **Main > Options > Edit Preferences** menu selection to open the Preferences dialog box shown below.

Use the Apply button to set temporary defaults for the current simulation. Use the Save button to write the preferences as permanent defaults to *modelsim.tcl*. Use the Close button to close the dialog box and return to the Main window (7-113) without making changes.

You can also use the **Main > Options > Save Preferences** menu selection to save current window settings to a tcl preference file.

The Preferences dialog box allows you to make preference changes **By Window** or **By Name** as shown below.

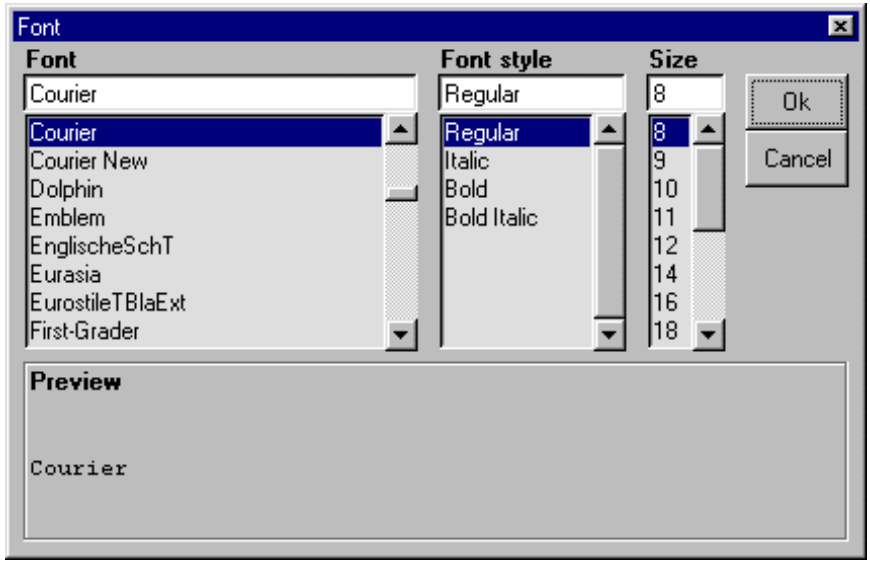
By Window page




The **By Window** page includes these options:

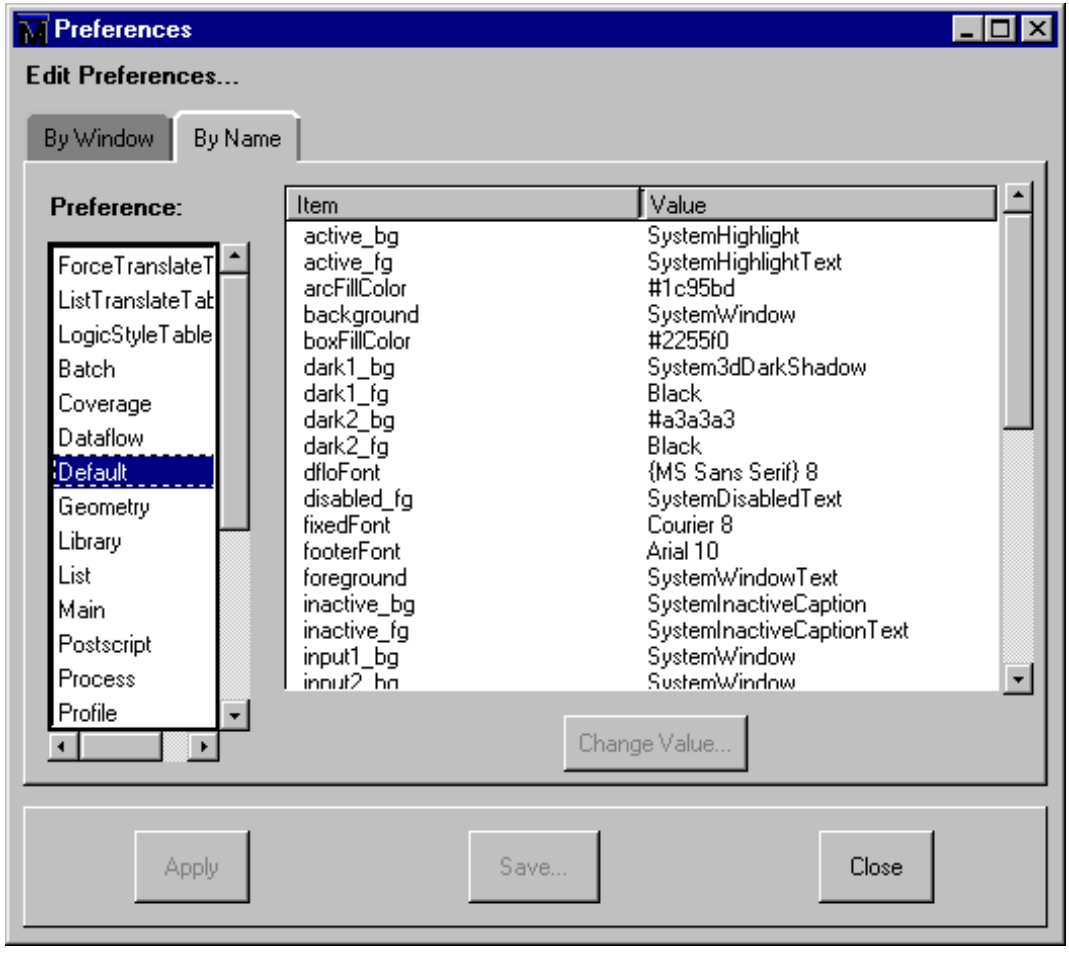
- **Window**
  Select the window type to modify; the color and font changes you make apply to all windows of this type. The Source window view allows you to preview source examples for either VHDL or Verilog. When you select the Source window, you can change preferences based on VHDL or Verilog source.
- **Colors**
  Select the color element to change, and choose a color from the palette; view your changes in the sample graphic at the center of the window.

- **Font**
  Select the Choose button; the Font Selection dialog box opens for your selection.




In the Font Selection dialog box, any selection you make automatically updates the By Window page; the **Reset** button scans your system for fonts and **Quit** closes the selection box.

By Name page




The **By Name** page includes these options:

- **Preference**
  Select the Preference and Item to change, and then click the **Change Value** button. Enter the new value into the field provided in the resulting dialog box.

In addition to window preferences (listed by window name in the Preference list), you may also set:

- **Default**
  Set default colors and fonts for menus and tree windows, also fill colors for VHDL (box) and Verilog (arc) structure symbols; these may be changed for individual windows.
- **Geometry**
  Set the default size and position for the selected window type; used for the geometry of any newly-created window. This option will only exist after preferences have been saved for the first time (using the Save button).
- **ForceTranslateTable**
  Specify how a string of digits are mapped into enumerations. This mapping is used for vectors only.
- **Library**
  Set colors for design libraries and library elements: architectures, configurations, entities, modules, and packages.
- **ListTranslateTable**
  Specify how various enumeration types map to the Model*Sim*'s internal logic types. This mapping is used for vectors only.
- **LogicStyleTable**
  Set the line type, color, and vertical location for internal logic types.
- **Postscript**
  Specifies postscript font mapping. The fonts you specify for the Dataflow and Wave windows on the By Window page are mapped to these fonts when the Dataflow/Wave window is output to postscript. See "Saving the Dataflow window as a Postscript file" (7-127) or "Saving waveforms" (7-189) for information on postscript output.
- **Startup**
  Set the location (geometry) of the Load a Design dialog box.
- **Vsim**
  This preference provides a method for attaching add-ons to Model*Sim*. This item must be set prior to simulation. If it is set from this dialog box, it will take effect the next time VSIM is invoked from the command line.
- **Welcome**
  Identifies the file which contains the text for the ModelSim QuickStart. **Do not change this item**.

The By Name page is a graphic representation of the "Preference variable arrays" (A-272) located in the *modelsim.tcl* preference file.

The changes you make in the Preferences dialog box are temporary for the current simulation. Your changes can be saved as permanent defaults by using **Main > Options > Save Preferences**. The new settings are saved to the current directory in the *modelsim.tcl* file by default. You may choose a different name for the TCL file if you wish.

See "Preference variable arrays" (A-272) for a list of the standard preference variables to be found in the *modelsim.tcl* file.

### Directly editing preference files

You can also modify variables by editing a preference file with any text editor. Make sure the file is saved as plain text. The Model*Sim* **notepad** (CR-70) provides a convenient way to edit a text file. At the Model*Sim* prompt type:

```
notepad <preference file path>
```

To edit the file once opened, change the read-only status with the Edit > read only menu selection.

**Note:** Case is significant in variable names, be careful when you edit variables!

## More preferences

Additional compiler and simulator preferences may be set in the *modelsim.ini* and MPF files, see "Preference variables located in INI and MPF files" (A-259).

## Preference variable loading order

Model*Sim* *.tcl*, INI, and MPF files all contain variables that are loaded when you start Model*Sim*. The files are evaluated for variable settings in the order below.

### .tcl file variables are evaluated before the design is loaded

Model*Sim* evaluates *.tcl* files prior to loading a design for simulation.

1. *The <install_dir>/modeltech/tcl/vsim/pref.tcl* file is always loaded.
2. The file specified by the MODELSIM_TCL (A-256) environment variable is loaded next,
3. if MODELSIM_TCL does not exist, the *modelsim.tcl* in the current directory is evaluated,
4. or if MODELSIM_TCL and *./modelsim.tcl* do not exist, the file specified by the HOME (A-255) environment variable used.

### INI and MPF file variables are evaluated after the design is loaded

After the design is loaded, *INI* or MPF file variables are found in these locations:

1. First the location specified by the MODELSIM (A-256) environment variable,

   If no MODELSIM variable exists, Mod*elSi*m looks for MPF and INI files in the locations shown below. Project files (MPF) are evaluated first, if no project file is found, Model*Sim* looks for a INI file in the same location.
2. next in the current directory if no MODELSIM variable exists,
3. then in the directory where the executable exists (*/install_dir/modeltech/<platform>*),
4. finally in the parent of the directory where the executable is (*/install_dir/modeltech*).

**Note:** The MODELSIM variable is generally set to an INI file. Setting the variable to a MPF file is not recommended since the file would contain project-specific information. Setting the MODELSIM variable to a MPF file is only recommended for batch-mode usage.

## Simulator state variables

Unlike other variables that must be explicitly set, simulator state variables return a value relative to the current simulation. Simulator state variables can be useful in commands, especially when used within a Model*Sim* DO files (macros).

| Variable | Result |
|---|---|
| argc | returns the total number of parameters passed to the current macro |
| architecture | returns the name of the top-level architecture currently being simulated; for a configuration or Verilog module, this variable returns an empty string |
| configuration | returns the name of the top-level configuration currently being simulated; returns an empty string if no configuration |
| delta | returns the number of the current simulator iteration |
| entity | returns the name of the top-level VHDL entity or Verilog module currently being simulated |
| library | returns the library name for the current region |
| MacroNestingLevel | returns the current depth of macro call nesting |
| n | represents a macro parameter, where n can be an integer in the range 1-9 |
| Now | returns the current simulation time expressed in the current time resolution, i.e., 1000 ns |
| now | returns the current simulation time as an absolute number of time steps, i.e., 1000 |
| resolution | returns the current simulation time resolution |

### Referencing simulator state variables

Variable values may be referenced in simulator commands by preceding the variable name with a $ sign. For example, to use the **now** and **resolution** variables in an **echo** command type:

```
echo "The time is $now $resolution."
```

Depending on the current simulator state, this command could result in:

```
The time is 12390 10ps.
```

If you do not want the dollar sign to denote a simulator variable, precede it with a "\". For example, \$now will not be interpreted as the current simulator time.

# B - Model*Sim* Shortcuts

## Appendix contents


Wave window keyboard shortcuts . . . . . 285

List window keyboard shortcuts . . . . . 286

Command shortcuts . . . . . 286

Command history shortcuts . . . . . 287

Mouse and keyboard shortcuts in the Transcript and Source windows . . 287

Right mouse button . . . . . 288


This appendix is a collection of the keyboard and command shortcuts available in the Model*Sim* GUI.

## Wave window keyboard shortcuts

Using the following keys when the mouse cursor is within the Wave window will cause the indicated actions:

| Key | Action |
|---|---|
| i I or + | zoom in |
| o O or - | zoom out |
| f or F | zoom full |
| l or L | zoom last |
| r or R | zoom range |
| <arrow up> | scroll waveform display up |
| <arrow down> | scroll waveform display down |
| <arrow left> | scroll waveform display left |
| <arrow right> | scroll waveform display right |
| <page up> | scroll waveform display up by page |

| Key | Action |
|---|---|
| <page down> | scroll waveform display down by page |
| <tab> | searches forward (right) to the next transition on the selected signal - finds the next edge |
| <shift-tab> | searches backward (left) to the previous transition on the selected signal - finds the previous edge |
| <control-f> | opens the find dialog box; search within the specified field in the wave-name pane for text strings |

## List window keyboard shortcuts

Using the following keys when the mouse cursor is within the List window will cause the indicated actions:

| Key | Action |
|---|---|
| <arrow up> | scroll listing up |
| <arrow down> | scroll listing down |
| <arrow left> | scroll listing left |
| <arrow right> | scroll listing right |
| <page up> | scroll listing up by page |
| <page down> | scroll listing down by page |
| <tab> | searches forward (down) to the next transition on the selected signal |
| <shift-tab> | searches backward (up) to the previous transition on the selected signal (does not function on HP workstations) |
| <control-f> | opens the find dialog box; find the specified item label within the list display |

## Command shortcuts

You may abbreviate command syntax, but there's a catch. The minimum characters required to execute a command are those that make it unique. Remember, as we add new commands some of the old shortcuts may not work.

## Command history shortcuts

The simulator command history may be reviewed, or commands may be reused, with these shortcuts at the Model*Sim*/VSIM prompt:

| Shortcut | Description |
|---|---|
| up and down arrows | scroll through the command history with the keyboard arrows |
| click on prompt | left-click once on a previous ModelSim or VSIM prompt in the transcript to copy the command typed at that prompt to the active cursor |
| `history` | shows the last few commands (up to 50 are kept) |

## Mouse and keyboard shortcuts in the Transcript and Source windows

The following mouse actions and special keystrokes can be used to edit commands in the entry region of the Main window. They can also be used in editing the file displayed in the Source window and all Notepad windows (enter the **notepad** command within Model*Sim* to open the Notepad editor).

| Mouse | Result |
|---|---|
| < left-button - click > | move the insertion cursor |
| < left-button - press > + drag | select |
| < shift - left-button - press > | extend selection |
| < left-button - double-click > | select word |
| < left-button - double-click > + drag | select word + word |
| < control - left-button - click > | move insertion cursor without changing the selection |
| < left-button - click > on previous ModelSim or VSIM prompt | copy and paste previous command string to current prompt |

| Keystrokes | Result |
|---|---|
| < left \| right - arrow > | move the insertion cursor |
| < up \| down - arrow > | scroll through command history |
| < control - p > | move insertion cursor to previous line |
| < control - n > | move insertion cursor to next line |
| < control - f > | move insertion cursor forward |
| < control - b > | move insertion cursor backward |
| < backspace > | delete character to the left |
| < control - d >, <Delete> | delete character to the right |
| < control - k > | delete to the end of line |
| < control - a >, <Home> | move insertion cursor to beginning of line |
| < control - e >, <Home> | move insertion cursor to end of line |
| < control - x ><br>< control - c ><br>< control - v > | cut selection |
| | copy selection |
| | insert clipboard |

The Main window allows insertions or pastes only after the prompt, therefore, you don't need to set the cursor when copying strings to the command line.

## Right mouse button

The right mouse button provides shortcut menus in the Main and Wave windows. In the Source window, the button gives you feedback on any HDL item under the cursor. See *Chapter 10 - ModelSim Graphic Interface* for menu descriptions.

# C - Tips and Techniques

## Appendix contents


Running command-line and batch-mode simulations . . . . . 290

Passing parameters to macros . . . . . 291

Source code security and -nodebug . . . . . 292

Setting up libraries for group use . . . . . 292

Detecting infinite zero-delay loops . . . . . 293

Modeling memory in VHDL . . . . . 293


This appendix is an effort to organize information to make it more accessible. We've collected documentation from several parts of the manual; some examples have evolved from answers to questions received by tech support. Your suggestions, tips, and techniques for this section would be appreciated.

## Running command-line and batch-mode simulations

The typical method of running Model*Sim* is interactive: you push buttons and/or pull down menus in a series of windows in the GUI (graphic user interface). But there are really three specific modes of **vsim** operation: GUI, command line, and batch. Here are their characteristics:

- **GUI mode**
  This is the usual interactive mode; it has graphical windows, push-button menus, and a command line in the text window. This is the default mode when **vsim** is invoked from within Model*Sim*.
- **Command-line mode - running vsim.exe**
  This an operational mode that has only an interactive command line; no interactive windows are opened. To run **vsim** in this manner, invoke it from the DOS prompt in Windows 95 or Windows NT.

  The resulting transcript file is created in such a way that the transcript can be re-executed without change if you desire. Everything except the explicit commands you enter will begin with a leading comment character (#).
- **Batch mode - running vsim.exe**
  Batch mode is an operational mode where no user interaction with the simulation is required. In this case, **vsim** is run from a Windows 95 or NT DOS prompt and standard input and output are re-directed to and from files. This works great for regression testing. An example is:

```
C:\modeltech> vsim ent arch <infile >outfile
```

where *infile* contains:

```
force reset 0
force clk 0, 0 1 50 -rep 100
run 10000
```

## Passing parameters to macros

In Model*Sim*, you invoke macros with the do command:

### Syntax

```
do
      <filename>  [ <parameter_value> ...]
```

### Arguments

`<filename>`
Specifies the name of the macro file to be executed.

`<parameter_value>`
Specifies values that are to be passed to the corresponding parameters $1 through $9 in the macro file. Multiple parameter values must be separated by spaces. If you specify fewer parameter values than the number of parameters used in the macro, the unspecified values are treated as empty strings in the macro.

There is no limit on the number of parameters that can be passed to macros, but only nine values are visible at one time. You can use the **shift** command (CR-90) to see the other parameters.

### See also

The **do** command (CR-43) for more information on do files.

## Source code security and -nodebug

The **-nodebug** option on both **vcom** (CR-106) and **vlog** (CR-141) hides internal model data. This allows a model supplier to provide pre-compiled libraries without providing source code and without revealing internal model variables and structure.

**Note:** Model*Sim*'s **-nodebug** compiler option provides protection for proprietary model information. The Verilog protect compiler directive provides similar protection, but uses a Cadence encryption algorithm that is unavailable to Model Technology.

If a design unit is compiled with **-nodebug** the Source window will not display the design unit's source code, the Structure window will not display the internal structure, the Signals window will not display internal signals (it still displays ports), the Process window will not display internal processes, and the Variables window will not display internal variables. In addition, none of the hidden objects may be accessed through the Dataflow window or with Model*Sim* commands.

Even with the data hiding of **-nodebug**, there remains some visibility into models compiled with **-nodebug**. The names of all design units comprising your model are visible in the library, and you may invoke **vsim** (CR-148) directly on any of these design units and see the ports. For this reason it is important to compile all design units with **-nodebug**.

Design units or modules compiled with -nodebug can only instantiate design units or modules that are also compiled -nodebug.

**Note:** For Verilog designs there is a small window of time - just after loading the design and before simulation begins - that a model's internals are available to the PLI access routines.

## Setting up libraries for group use

By adding an "others" clause to your *modelsim.ini* file, you can have a hierarchy of library mappings. If the Model*Sim* tools don't find a mapping in the *modelsim.ini* file, then they will search the library section of the initialization file specified by the "others" clause. For example:

```
[library]
asic_lib = /cae/asic_lib
work = my_work
others = /usr/modeltech/modelsim.ini
```

## Detecting infinite zero-delay loops

VHDL simulation uses steps that advance simulated time, and steps that do not advance simulated time. Steps that do not advance simulated time are called "delta cycles". Delta cycles are used when signal assignments are made with zero time delay.

If a large number of delta cycles occur without advancing time, it is usually a symptom of an infinite zero-delay loop in the design. In order to detect the presence of these loops, Model*Sim* defines a limit, the “iteration_limit", on the number of successive delta cycles that can occur. When the iteration_limit is exceeded, **vsim** stops the simulation and gives a warning message.

You can set the iteration_limit from the **Simulation > Properties** menu, or by modifying the *modelsim.ini* file.

The iteration_limit default value is 1000.

When you get an iteration_limit warning, first increase the iteration limit and try to continue simulation. If the problem persists, look for zero-delay loops.

One approach to finding zero-delay loops is to increase the iteration limit again and start single stepping. You should be able to see the assignment statements or processes that are looping. Looking at the Process window will also help you to see the active looping processes.

When the loop is found, you will need to change the design to eliminate the unstable loop.

See "Projects and system initialization" (3-35) for more information on modifying the *modelsim.ini* file.

## Modeling memory in VHDL

As a VHDL user, you might be tempted to model a memory using signals. Two common simulator problems are the likely result:

- You may get a "memory allocation error" message, which typically means the simulator ran out of memory and failed to allocate more storage.
- Or, you may get very long load, elaboration or run times.

These problems are usually explained by the fact that signals consume a substantial amount of memory (many dozens of bytes per bit), all of which needs to be loaded or initialized before your simulation starts.

A simple alternative implementation provides some excellent performance benefits:

- storage required to model the memory can be reduced by 1-2 orders of magnitude
- startup and run times are reduced
- associated memory allocation errors are eliminated

The trick is to model memory using variables instead of signals.

In the example below, we illustrate three alternative architectures for entity "memory". Architecture "style_87_bad" uses a vhdl signal to store the ram data. Architecture "style_87" uses variables in the "memory" process, and architecture "style_93" uses variables in the architecture.

For large memories, architecture "style_87_bad" runs many times longer than the other two, and uses much more memory. This style should be avoided.

Both architectures "style_87" and "style_93" work with equal efficiently. You'll find some additional flexibility with the VHDL 1993 style, however, because the ram storage can be shared between multiple processes. For example, a second process is shown that initializes the memory; you could add other processes to create a multi-ported memory.

To implement this model, you will need functions that convert vectors to integers. To use it you will probably need to convert integers to vectors.

Example functions are provided below in package "conversions".

```
------------------------------------------------------------
------------------------------------------------------------
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;
use work.conversions.all;

entity memory is
        generic(add_bits : integer := 12;
                data_bits : integer := 32);
        port(add_in : in std_ulogic_vector(add_bits-1 downto 0);
            data_in : in std_ulogic_vector(data_bits-1 downto 0);
            data_out : out std_ulogic_vector
                               (data_bits-1 downto 0);
            cs, mwrite : in std_ulogic;
            do_init : in std_ulogic);
        subtype word is std_ulogic_vector(data_bits-1 downto 0);
        constant nwords : integer := 2 ** add_bits;
        type ram_type is array(0 to nwords-1) of word;

end;
```

```
architecture style_93 of memory is
        -----------------------------
        shared variable ram : ram_type;
        -----------------------------
begin
memory:
process (cs)
        variable address : natural;
        begin
            if rising_edge(cs) then
                address := sulv_to_natural(add_in);
                if (mwrite = '1') then
                    ram(address) := data_in;
                    data_out <= ram(address);
                else
                    data_out <= ram(address);
                end if;
            end if;
        end process memory;
-- illustrates a second process using the shared variable
initialize:
process (do_init)
        variable address : natural;
        begin
            if rising_edge(do_init) then
                for address in 0 to nwords-1 loop
                    ram(address) := data_in;
                end loop;
            end if;
        end process initialize;
end architecture style_93;



architecture style_87 of memory is
begin
memory:
process (cs)
        ----------------------
        variable ram : ram_type;
        ----------------------
        variable address : natural;
        begin
            if rising_edge(cs) then
                address := sulv_to_natural(add_in);
                if (mwrite = '1') then
                    ram(address) := data_in;
```

```
                data_out <= ram(address);
            else
                data_out <= ram(address);
            end if;
        end if;
    end process;
end style_87;



architecture bad_style_87 of memory is
    ----------------------
    signal ram : ram_type;
    ----------------------
begin
memory:
process (cs)
    variable address : natural := 0;
    begin
        if rising_edge(cs) then
            address := sulv_to_natural(add_in);
            if (mwrite = '1') then
                ram(address) <= data_in;
                data_out <= data_in;
            else
                data_out <= ram(address);
            end if;
        end if;
    end process;
end bad_style_87;



------------------------------------------------------------
------------------------------------------------------------
use std.standard.all;
library ieee;
use ieee.std_logic_1164.all;

package conversions is
    function sulv_to_natural(x : std_ulogic_vector) return
                natural;
    function natural_to_sulv(n, bits : natural) return
                std_ulogic_vector;
end conversions;

package body conversions is

    function sulv_to_natural(x : std_ulogic_vector) return
                natural is
```

```
            variable n : natural := 0;
            variable failure : boolean := false;
        begin
            assert (x'high - x'low + 1) <= 31
                report "Range of sulv_to_natural argument exceeds
                    natural range"
                severity error;
            for i in x'range loop
                n := n * 2;
                case x(i) is
                    when '1' | 'H' => n := n + 1;
                    when '0' | 'L' => null;
                    when others    => failure := true;
                end case;
            end loop;
            assert not failure
                report "sulv_to_natural cannot convert indefinite
                    std_ulogic_vector"
                severity error;


            if failure then
                return 0;
            else
                return n;
            end if;
        end sulv_to_natural;

        function natural_to_sulv(n, bits : natural) return
                    std_ulogic_vector is
            variable x : std_ulogic_vector(bits-1 downto 0) :=
                    (others => '0');
            variable tempn : natural := n;
        begin
            for i in x'reverse_range loop
                if (tempn mod 2) = 1 then
                    x(i) := '1';
                end if;
                tempn := tempn / 2;
            end loop;
            return x;
        end natural_to_sulv;

end conversions;
```

# Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

## A

architecture simulator state variable 283
argc simulator state variable 283
AssertFile .ini file variable 262
AssertionFormat .ini file variable 263
Assertions
    selecting severity that stops simulation 213

## B

Batch mode 290
Break
    on assertion 213
BreakOnAssertion .ini file variable 263
Breakpoints
    deleting 152
    setting 152
    viewing 152

## C

Cell libraries 71
CheckpointCompressMode .ini file variable 263
CheckSynthesis .ini file variable 261
Colors
    changing window colors with the GUI 277
Command reference 20
CommandHistory .ini file variable 263
Command-line mode 290
Commands
    library management commands 25
    VSIM Tcl commands 250
Compilation and Simulation
    Verilog 54–94
    VHDL 43–52
Compiler directives 78
    IEEE Std 1364-1995 78
    XL compatible compiler directives 79
Compiling
    locating source errors 196
    projects 41
    Verilog
        XL 'uselib compiler directive 63
        XL compatible options 60
    Verilog designs 55
        incremental compilation 56
    with VITAL packages 52
compiling
    with the graphic interface 195
concatenation
    of signals 104
ConcurrentFileLimit .ini file variable 263
configuration simulator state variable 283
Cur_Top_DUs .mpf file variable 266

## D

Dataset Browser 97
Datasets
    restrict dataset prefix display 99
    the Dataset Browser 97
DefaultForceKind .ini file variable 263
DefaultRadix .ini file variable 263
Delay
    specifying stimulus delay 149
DelayFileOpen .ini file variable 264
Delta
    collapse deltas in the List window 132
    referencing simulator iteration
        as a simulator state variable 283
delta simulator state variable 283
Dependent design units 45
Descriptions of HDL items 156
Design hierarchy
    viewing in Structure window 158
Design library

assigning a logical name 29
creating 25
for VHDL design units 44
mapping search rules 31
resource type 24
working type 24
Design units 24
viewing hierarchy 111
Directories
moving libraries 31
See also, Libraries
DOPATH environment variable 255
DOPATH simulator control variable 275

## E

Editing
in notepad windows 121, 287
in the Main window 121, 287
in the Source window 121, 287
EDITOR environment variable 255
entity simulator state variable 283
Environment variables 255
in PrefMain(file) variable 274
location of modelsim.ini file 282
overriding with DOPATH simulator control variable 275
referencing from ModelSim command line 258
referencing with VHDL FILE variable 257
setting before compiling or simulating 255
setting in Windows 256
specify transcript file location with TranscriptFile 265
specifying library locations in modelsim.ini file 259
variable substitution using Tcl 248
Errors during compilation, locating 196
exiting ModelSim
turning off confirmation dialog 274
Explicit .ini file variable 261

## F

file types
.ini and .mpf 36
Finding
a cursor in the Wave window 185
a marker in the List window 140
Finding names, and searching for values in windows 110
Fonts
changing fonts with the GUI 278
Foreign language interface
tracing 93

## G

GenerateFormat .ini file variable 264
Graphic interface 105–??

## H

Hazard .ini file variable (VCOM) 261
Hazard .ini file variable (VLOG) 262
History shortcuts 287
HOME environment variable 255
how_Warning5 .ini file variable 260

## I

ieee .ini file variable 259
IEEE Standard 1076 18, 43
IEEE Standard 1364 18, 54
IgnoreError .ini file variable 264
IgnoreFailure .ini file variable 264
IgnoreNote .ini file variable 264
IgnoreVitalErrors .ini file variable 261
IgnoreWarning .ini file variable 264
incremental compilation
automatic 58
manual 57
Incremental compilation, Verilog 56

Initialization file, see Project files
Instantiation label 159
Iteration_limit
    detecting infinite zero-delay loops 293
IterationLimit .ini file variable 264

## K

Keyboard shortcuts
    List window 140, 286
    Wave window 187, 285

## L

<Lib>_script .mpf file variable 266
Libraries
    alternate IEEE libraries 33
    creating design libraries 25
    deleting library contents 27
    design library types 24
    design units 24
    ieee_numeric 33
    ieee_synopsis 33
    library management commands 25
    mapping from the command line 30
    mapping hierarchy 268
    mapping search rules 31
    mapping with the GUI 29
    moving 31
    naming 29
    predefined 32
    rebuilding ieee_numeric 33
    rebuilding ieee_synopsis 33
    refreshing library images 33
    resource libraries 24
    setting up for groups 292
    std 32
    verilog 58
    VHDL library clause 32
    viewing library contents 27
    working libraries 24
library simulator state variable 283
License .ini file variable 264
List window
    setting triggers 131
List window (see also, Windows) 128
LM_LICENSE_FILE environment variable 255
Locating source errors during compilation 196
logfiles 95

## M

MacroNestingLevel simulator state variable 283
Macros (do files)
    depth of nesting, simulator state variable 283
    parameter as a simulator state variable (n) 283
    parameter total as a simulator state variable 283
    passing parameters to 291
    startup macros 268
Main window (see also, Windows) 113
Memory
    modeling in VHDL 293
Menus
    Dataflow window 125
    List window 129
    Main window 114
    Process window 143
    Signals window 146
    Source window 153
    Structure window 159
    tearing off or pinning menus 110
    Variables window 162
    Wave window 169
Messages
    turning off assertion messages 269
    turning off warnings from arithmetic packages 269
MGC_LOCATION_MAP environment variable 255
MODEL_TECH environment variable 255
MODEL_TECH_TCL environment variable 255
Modeling memory in VHDL 293
MODELSIM environment variable 256
modelsim.ini, see Project files

MODELSIM_TCL environment variable 256
MPF file 36
MTI_TF_LIMIT environment variable 255
Multiple drivers on unresolved signal 199

## N

n simulator state variable 283
Nets
    adding to the Wave and List windows 149
    displaying values in Signals window 144
    forcing signal and net values 147
    saving values as binary log file 149
    viewing waveforms 164
Next and previous edges, finding 187, 285
No space in time literal 198
NoDebug .ini file variable (VCOM) 261
NoDebug .ini file variable (VLOG) 262
Notepad windows, text editing 121, 287
NoVital .ini file variable 261
NoVitalCheck .ini file variable 261
Now simulator state variable 283
now simulator state variable 283
NumericStdNoWarnings .ini file variable 264

## O

Optimize for std_logic_1164 199
Optimize_1164 .ini file variable 260

## P

Packages
    standard 32
    textio 32
Parameters, for macros 291
PathSeparator .ini file variable 264
PLI see Verilog PLI
PLIOBJS environment variable 255
PlotFilterResolution simulator control variable 275
Preferences
    setting preferences with the GUI
        window fonts and colors 276
        window sizes and locations 276
Process window (see also, Windows) 142
Process without a wait statement 198
Processes
    values and pathnames in Variables window 161
Project files
    compile a project 41
    creating a project 38
    modelsim.ini
        default to VHDL93 269
        environment variables 267
        hierarchial library mapping 268
        opening VHDL files 270
        to specify a startup file 268
        turning off arithmetic warnings 269
        turning off assertion messages 269
        using to create a transcript file 268
        using to define force command default 269
        using to delay file opening 270
    modelsim.mpf 35
        project definition 36
    modifying a project 41
    open a project 41
    project operations 37
    simulating a project 41
’protect compiler directive 292

## Q

Quick Start guide 214
Quiet .ini file variable (VCOM) 261
Quiet .ini file variable (VLOG) 262
quitting ModelSim
    turning off confirmation dialog 274

## R

Radix

specifying in List window 136
specifying in Signals window 148
reconstruct RTL-level design busses 101
Records
changing values of 161
Refreshing library images 33
Register variables
adding to the Wave and List windows 149
displaying values in Signals window 144
saving values as binary log file 149
viewing waveforms 164
Resolution .ini file variable 265
resolution simulator state variable 283
resource library 24
restart 116, 120
RunLength .ini file variable 265

## S

ScalarOpts .ini file variable 261, 262
SDF
Errors and warnings 219
Instance specification 218
interconnect delays 229
mixed VHDL and Verilog designs 229
troubleshooting 230
Verilig
rounded timing values 229
Verilog
$sdf_annotate system task 222
optional conditions 228
optional edge specifications 227
SDF to Verilog construct matching 223
Verilog SDF annotation 222
VHDL
Resolving errors 221
SDF to VHDL generic matching 220
Searching
for values and finding names in windows 110
List window
signal values, transitions, and names 139
text strings in the List window 139
text strings in the Wave window 183
Verilog libraries 59
waveform
signal values, edges and names 183
searchLog simulator command 103
Shortcuts
command history 287
command line caveat 286
List window 140, 286
text editing 121, 287
Wave window 187, 285
Show source lines with errors 198
Show_source .ini file variable (VCOM) 260
Show_source .ini file variable (VLOG) 262
Show_VitalChecksWarning .ini file variable 260
Show_Warning1 .ini file variable 260
Show_Warning2 .ini file variable 260
Show_Warning3 .ini file variable 260
Show_Warning4 .ini file variable 260
Signal transitions
searching for 186
Signals
adding to a log file 149
adding to the Wave and List windows 149
applying stimulus to 147
combining into a user-defined bus 110
displaying values in Signals window 144
forcing signal and net values 147
saving values as binary log file 149
selecting signal types to view 147
viewing waveforms 164
Signals window (see also, Windows) 144
Simulating
applying stimulus
see also VSIM command, force
applying stimulus to signals and nets 147
applying stimulus with textio 50
command-line mode 290
projects 41
saving waveform as a Postscript file 189
setting default run length 212

setting iteration limit 212
setting time resolution 203
Verilog
delay modes 71
even order issues 66
hazard detection 67
resolution limit 65
XL compatible simulator options 68
Verilog designs 65
VHDL designs 45
viewing results in List window 128
with VITAL packages 52
simulating
with the graphic interface 202
Simulation and Compilation
Verilog 54–94
VHDL 43–52
Sorting
sorting HDL items in VSIM windows 110
Source code
source code security 292
viewing 152
Source directory, setting from source window 153
SourceDir simulator control variable 275
SourceMap simulator control variable 275
Src_Files .mpf file variable 266
standards supported 18
Startup
macro in the modelsim.ini file 265
using a startup file 268
Startup .ini file variable 265
Startup macros 268
Status bar
Main window 121
std .ini file variable 259
std_developerskit .ini file variable 259
StdArithNoWarnings .ini file variable 265
STDOUT environment variable 256
Structure window (see also, Windows) 158
synopsys .ini file variable 260
System initialization 35


## T


Tcl 239–252
command separator 247
command substitution 246
evaluation order 247
Man Pages in Help menu 118
relational expression evaluation 248
variable substitution 234, 248
Tcl command syntax 242
text and command syntax 20
Text editing, see Editing
Text strings
finding in the List window 139
finding in the Wave window 183
TextIO package 43
alternative I/O files 50
containing hexadecimal numbers 49
dangling pointers 49
ENDFILE function 50
ENDLINE function 50
file declaration 47
implementation issues 48
providing stimulus 50
standard input 48
standard output 48
WRITE procedure 48
WRITE_STRING procedure 49
Time
setting the resolution 46, 65
time as a simulator state variable 283
time resolution as a simulator state variable 283
Tool bar
Main window 119
Wave window 172
Top_DUs .mpf file variable 266
Tracing HDL items with the Dataflow window 126
TranscriptFile .ini file variable 265
Tree windows
VHDL and Verilog items in 111
viewing the design hierarchy 112
Triggers, setting in the List window 131

## U

Unbound Component 198
UnbufferedOutput .ini file variable 265
UpCase .ini file variable 262
Use 1076-1993 language standard 197
Use clause
  specifying a library 32
Use explicit declarations only 197
user-defined buses 100
UserTimeUnit .ini file variable 265

## V

Values of HDL items 156
Variable settings report 267
Variables window (see also, Windows) 161
Variables, HDL
  changing value of with the GUI 161
Variables, referencing
  loading order at ModelSim startup 282
  simulator state variables
    iteration number 283
    name of entity or module as a variable 283
    resolution 283
    simulation time 283
Variables, setting
  environment variables 255
VCD files
  VCD system tasks 234
Verilog
  cell libraries 71
  compiler directives 78
  compiling design units 55
  compiling with XL 'uselib compiler directive 63
  creating a design library 55
  library usage 58
  SDF annotation 222
  sdf_annotate system task 222
  simulating 65
    delay modes 71
    event order issues 66
    XL compatible options 68
  simulation hazard detection 67
  simulation resolution limit 65
  source code viewing 152
  system tasks 72
  XL compatible compiler options 60
  XL compatible routines 92
  XL compatible system tasks 75
verilog .ini file variable 259
Verilog PLI 80–94
  specifying the PLI file to load 83
  support for VHDL objects 88
verilog standards 18
Veriuser .ini file variable 265
VHDL
  delay file opening 270
  Dependency checking 45
  file opening delay 270
  library clause 32
  object support in PLI 88
  simulating 45
  source code viewing 152
  timing check disabling 46
vhdl standards 18
VHDL93 .ini file variable 260
Viewing design hierarchy 111
virtual hide command 101
Virtual objects 100
  virtual functions 102
  virtual regions 102
  virtual signals 101
  virtual types 103
virtual region command 102
Virtual regions
  reconstruct the RTL Hierarchy in gate level design 102
virtual save command 101
virtual signal command 101
Virtual signals
  reconstruct RTL-level design busses 101
  reconstruct the original RTL hierarchy 101

virtual hide command 101
VITAL
compiling and simulating with accelerated VITAL packages 52
obtaining the specification and source code 51
VITAL packages 51
VSIM commands
searchLog 103


## W


Warnings
turning off warnings from arithmetic packages 269
Wave window (see also, Windows) 164
WaveSignalNameWidth .ini file variable 266
Windows
change window fonts and colors 276
finding HDL item names 110
opening with the GUI 116
save window sizes and locations 276
searching for HDL item values 110
Dataflow window
tracing signals and nets 126
List window 128
adding HDL items 134
adding signals with a log file 149
examining simulation results 138
formatting HDL items 135
locating time markers 110
saving to a file 141
setting display properties 131
Main window 113
status bar 121
text editing 121, 287
time and delta display 121
tool bar 119
Process window 142
displaying active processes 142
specifying next process to be executed 142
viewing processing in the region 142
Signals window 144
VHDL and Verilog items viewed in 144
Source window
text editing 121, 287
Structure window 158
HDL items viewed in 158
instance names 159
selecting items to view in Signals window 144
VHDL and Verilog items viewed in 158
viewing design hierarchy 158
Variables window 161
displaying values 161
VHDL and Verilog items viewed in 161
Wave window 164
adding HDL items 175
adding signals with a log file 149
changing display range (zoom) 186
cursor measurements 185
locating time cursors 110
setting display properties 182
using time cursors 184
zoom options 186
zooming 186
work library 24
Work_Libs .mpf file variable 266


## Z


Zero-delay loop, detecting infinite 293
Zoom
from Wave toolbar buttons 187
from Zoom menu 186
options 186
with the mouse 187